Processus flexible de configuration pour lignes de produits logiciels complexes

Simon Urli

UNIVERSITE DE NICE SOPHIA-ANTIPOLIS

ECOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THESE

pour l'obtention du grade de

Docteur en Sciences

de l'Université de Nice Sophia-Antipolis

Mention : **Informatique**

présentée et soutenue par

SIMON URLI

Processus Flexible de Configuration pour Lignes de Produits Logiciels Complexes

Thèse dirigée par *Mireille Blay-Fornarino*

soutenue le 26 février 2015

Jury :

Rapporteurs	Mme. Marianne HUCHARD	Professeur Université Montpellier 2
	M. Olivier BARAIS	Maître de Conférence HDR Université de Rennes 1
Examinateurs	M. Patrick HEYMANS	Professeur Université de Namur
	M. Michel RIVEILL	Professeur Université Nice Sophia Antipolis
Directrice	Mme. Mireille BLAY-FORNARINO	Professeur Université Nice Sophia Antipolis

REMERCIEMENTS

Le 5 janvier 2015,
TGV n°5164 Lille → Antibes

J'ai toujours considéré ma thèse comme un voyage, aussi bien scientifique que géographique. Mais ça a surtout été, et de manière bien plus importante, un voyage humain. Ainsi j'ai eu la chance de rencontrer durant ces trois ans et demi des personnes très différentes qui ont, chacune à leur manière, apporté leur pierre à cette thèse. C'est toutes ces personnes que je tiens à remercier ici – et je m'excuse par avance pour ceux que je vais nécessairement oublier – et c'est à eux et à toutes les autres personnes qui contribuent ou vont contribuer à me faire avancer que je dédie cette thèse.

Ces remerciement prendront donc la forme d'un processus flexible de configuration, ou encore d'un voyage dans un train de la SNCF, avec des changements... et des retards.

Les origines

Comme je l'ai dit, ce voyage de thèse a été également géographique pour moi : du Nord (celui au dessus de Paris, entendons-nous bien) vers le Sud (celui en dessous de Lyon). Je tiens avant tout à remercier *mes* gens du Nord. Mes parents en tout premier lieu, Hélène et Jean-Christophe, qui sont toujours là pour moi même quand j'oublie – trop souvent ! – de donner des nouvelles, et qui ont eu la patience quand j'étais petit de répondre à mes innombrables questions et surtout de me supporter faire mes expérimentations ! Merci ! Je ne serai pas là où j'en suis sans la petite grande sœur non plus, qui a expérimenté la thèse avant moi : tu gères, donc n'oublie pas de respirer :)

Je ne serai pas non plus allé très loin sans les amis de là haut, ceux d'il y a 10 ans ou moins, ceux qui ont complètement participé à ma construction et qui continuent aussi à me donner des baffes et des coups de pouces métaphoriques pour me donner de l'élan ! Merci Jue en premier, j'ai toujours cette photo débile sur ma porte, donc continue à m'envoyer des bouffées de légèreté quand je râle et à me donner le smile juste en me laissant un message. Et puis merci aussi Rémy pour avoir toujours été là, même maladroitement, et puis pour continuer à bouillonner et à me communiquer cette énergie, parfois malgré moi ! Je n'aurais pas tenu le coup non plus si tu n'avais pas été virtuellement là plein de fois aussi Clémentine, donc merci encore pour les innombrables conversations et remontages de moral :) Lye, déjà je ne serai jamais arrivé ici sans toi techniquement et merci encore d'être certainement le plus sain d'esprit de nous tous (ça en dit long sur la fine équipe...). François, je n'ai toujours pas compris comment tu fonctionnais vraiment, mais vu le nombre de fois où tu m'as sauvé la mise je ne regrette pas d'avoir squatté les mêmes bancs de fac, et merci encore pour les côtes de porc ;)

Et merci à vous tous aussi Céline, Marina, Sophie, Julie, Antoine, Clément et Grég !

Dans cet univers de gens super, j'ai eu la chance de faire mon stage au LIFL grâce à Raphaël Marvie et Xavier Le Pallec, donc merci à vous deux pour m'avoir donné l'occasion de mettre le

pied à l'étrier et pour les conseils qui m'ont mené là où j'en suis aujourd'hui !

Le labo

Et puis par une nuit de septembre 2011 j'ai pris la route pour Antibes. Mais ça ne serait jamais arrivé si Xavier ne m'avait pas encouragé à voir Sébastien Mosser et si lui même ne m'avait pas parlé d'un "petit projet bien cool sur Sophia", donc merci encore de m'avoir guidé cette fois là et par la suite également !

J'ai été dirigé et encadré pendant mes trois ans et quelques de thèse par la meilleure directrice de thèse – et certainement une des seules capable de sprinter en talon – imaginable, donc juste UN ENORME MERCI Mireille ! Et promis on organisera le concours de "Sprint de couloir" inter-équipe incessamment ;)

Apparté publicitaire :

Je tiens à remercier les compagnies productrices de sodas sucrés et caféinés à base de cola, de capsules de café et de biscuits fourrés au chocolat pour leur soutien logistique inébranlable quoique frayeux durant ces trois ans et demi de dur labeur.
Ces remerciements sont cependant à pondérer par la surcharge du même nom à laquelle ils ont pu hypothétiquement participer.

Mon arrivée au labo a été rythmée par un plafond de bureau qui s'est effondré ce qui est une méthode imparable pour briser la glace avec ses collègues. Donc merci à mes super co-bureau successifs : Javier, Filip, Nadia, Xheva et Cyril. Javier tu gagnes la palme du plus grand nombre de concerts avec un co-bureau et ils étaient vraiment cool en plus, donc merci de m'avoir fait découvrir la trop rare vie culturelle du secteur ! Filip tu gagnes sans conteste la palme du plus grand nombre de bières bues avec un co-bureau même si on n'a partagé le même bureau que très peu de temps durant LE GRAND DÉMÉNAGEMENT, et merci encore pour toute ton aide dans le setup Maven/Eclipse. Nadia tu gagnes la palme de l'aide LATEX la plus complète de l'histoire, ton setup m'a juste fait gagner des journées de boulot, merci ! Xheva, you certainly win the prize of the most patient office-mate I had ! Thanks ! Et Cyril tu gagnes la palme du co-bureau le plus officiellement dangereux, mais vu que tu supportes mes grognements de wookie, je t'excuse :)

Un merci spécial à Ivan qui a été une espèce de co-bureau malgré lui de par la minceur des cloisons, merci pour ta patience et pour ne pas trop dévoiler ton côté hippie/rainbow :p

J'évoque les personnes avec qui j'ai partagé le même bureau, mais je peux remercier aussi l'ensemble des doctorants que j'ai côtoyé : Yohann, Stéphanie, Emmanuel, Katy, Romaric, Tran, Christian D., Alban, André ; ainsi que les permanents du labo : Johan, Michel, Anne-Marie, Frédéric P., Fabien, Gaetan, Alain, Philippe L. sans oublier Sabine et Françoise ! Et un merci spécial à Christophe et Franck pour les soirées et les discussions !

YourCast

J'ai travaillé pendant ma thèse sur le projet YOURCAST qui m'a permis de collaborer avec des personnes vraiment géniales. Je tiens à remercier ici tout particulièrement Philippe Collet aka Philou pour toute l'aide que tu as pu m'apporter aussi bien sur les questions scientifiques, sur l'écriture des articles, que sur les opportunités de cours, merci ! Je dois également remercier

Philippe Renevier pour ton aide sur les IHM et pour m'avoir mis le pied à l'étrier pour les cours de COO. Et je tiens aussi à vous remercier tous les deux ensemble pour votre humour omniprésent : le labo serait bien plus morne sans vous (et on attend toujours les Phi-Stars !). Merci aussi à Laurence Duchien pour tous les conseils que tu as pu me donner pendant le projet et pour m'avoir fait confiance alors que j'étais loin d'être un élève modèle dans tes cours ! Et merci à Daniel Romero pour tout le boulot réalisé sur la génération ! Enfin merci à Philippe Salvan (encore un Philippe !) pour nous avoir donné l'opportunité d'expérimenter à Vaison et pour m'avoir permis de faire cette expérience chorale dans un théâtre antique !

La Thèse

Si ma thèse a été essentiellement encadrée par Mireille, que je remercie encore ici, je ne serai jamais arrivé au bout sans les aides de Philou et Séb, que je remercie encore également ici.

D'autres personnes ont également collaboré plus ou moins directement à mes travaux, que je tiens à remercier ici. I first have to thank Robert France for the incredibly enriching conversations we had at the beginning of my thesis and to really have helped me defining my concepts. Merci également à Claudine Peyrat et Frédéric Havet pour leur aide précieuse sur l'algorithme de réalisabilité ! J'ai aussi eu l'occasion de partir travailler au Chili pendant un mois en début de thèse, donc merci à Alexandre Bergel d'avoir accepté de m'accueillir et d'avoir proposé une autre vision de mes travaux.

Merci également à toutes les personnes que j'ai pu croiser en conférence et qui m'ont aidé d'une manière ou d'une autre à faire avancer mon travail.

Enfin, merci à Marianne Huchard et Olivier Barais d'avoir accepté de rapporter cette thèse, et merci encore pour toutes vos remarques qui m'ont permis d'améliorer énormément de choses dans ce manuscrit ! Merci également à Patrick Heymans d'avoir accepté de faire parti de mon jury malgré un emploi du temps extrêmement rempli ! Enfin merci Michel Riveill, à la fois pour avoir accepté également de faire parti de mon jury malgré toutes les responsabilités que tu as, mais également pour nous avoir énormément aidé pour le projet et pour tes conseils pour la startup !

Et ensuite ?

Je n'ai pas encore eu l'occasion de remercier toutes les personnes que j'ai pu rencontrer sur la côte d'azur et qui, si elles n'ont pas directement participé à cette thèse, m'ont permis personnellement d'avancer ! Merci, tout d'abord à Alice, Céline, Marion, Caro et Elise, pour avoir été là.

Ensuite merci à Flo : tu fais certainement partie des personnes les plus intéressantes que j'ai rencontrée donc ne lâche rien ! :) Merci à Lucy aussi : continue de m'envoyer de l'énergie, même si tu es loin maintenant ! Merci à Mathias, parce que discuter des mérites comparés des systèmes multi-agents et des équations de dynamiques des fluides après quelques – trop de – pintes ça n'a pas de prix !

Il y en a un que je n'ai pas encore remercié et il va probablement lire ça un jour et savoir que je parle de lui. Donc merci Christian pour les discussions depuis le début de ma thèse, pour ton optimisme inébranlable et pour accepter de monter une boîte avec le doux-dingue que je suis ! Et si les écrans ne marchent pas, on se lance dans la création de Punching-Ball !

Et pour mettre un point final à ces remerciements :

Merci à vous de me lire.

TABLE DES MATIÈRES

CHAPITRE 1

INTRODUCTION

Sommaire

1.1 Contexte et Motivation

Nous sommes aujourd'hui dans une société numérique. Les logiciels informatiques font partie de notre quotidien et nous les utilisons aussi bien volontairement, qu'involontairement à travers les nombreux systèmes que nous côtoyons sans même nous en apercevoir. Il convient donc d'être capable de produire de plus en plus vite des logiciels de qualité pour des systèmes de plus en plus complexes. Dans ce contexte, nous proposons une nouvelle approche de la production des logiciels complexes basée sur les lignes de produits logiciels.

1.1.1 Système-de-Systèmes

Les problématiques de l'ingénierie logicielle ne concernent plus seulement les techniques de réalisation d'un système unique, mais bien désormais l'utilisation de manière cohérente d'un ensemble de systèmes à travers des infrastructures de grande taille [Northrop 06] : on parle dans ce cadre de *système-de-systèmes*.

Ces système-de-systèmes ont entre autres caractéristiques d'être composés de systèmes indépendants, d'être géographiquement distribués et de définir des comportements émergents [Maier 98, Boardman 06]. Ces nouveaux systèmes sont complexes à réaliser, à la fois de part leurs dimensions, mais également de part la difficulté de s'assurer de la cohérence des assemblages de systèmes. Cette thèse propose donc de s'appuyer sur le paradigme des lignes de produits logiciels afin de faciliter la production de ce type de systèmes.

1.1.2 Lignes de Produits Logiciels : vers la réutilisation de masse

Les besoins en terme de développements logiciels ont forcé l'industrie à se tourner vers des solutions permettant une réutilisation massive de codes logiciels déjà existants. L'amélioration des langages de programmation [Djisktra 76], la définition de nouveaux paradigmes comme la programmation orientée aspect [Kiczales 97] ou composant [Nierstrasz 92], ou encore la définition de nouvelles architectures comme les architectures orientées services [Helferich 07] ont eu pour rôles de faciliter la séparation des préoccupations et de favoriser la réutilisation.

Force est de constater que les logiciels sont essentiellement aujourd'hui le fruit d'un assemblage judicieux de codes réutilisés. Dès 1976 David Parnas évoquait des famille de programmes [Parnas 76] pour désigner des ensembles de logiciels partageant des caractéristiques communes et des variantes qu'il est nécessaire de déterminer à l'avance afin de pouvoir optimiser la réalisation des logiciels. Cette notion a plus tard été formalisée par la définition des *Lignes de Produits Logiciels* (LPL) qui font l'objet des travaux de cette thèse.

Les LPL que nous étudions ont donc pour objectif de maximiser la réutilisation de codes existants dans le cadre d'une famille de produits afin de réduire les coûts de production des logiciels [Gaffney 92] mais également afin de garantir leur cohérence.

1.1.3 Des LPL pour les systèmes-de-systèmes : vers la composition de masse

De fait, de nombreuses LPL ont été réalisées afin de produire massivement des logiciels, en particulier pour les intégrer dans du matériel [Clements 02]. Or l'exploitation des LPL pour la réalisation de systèmes-de-systèmes pose de nouvelles problématiques. Il ne s'agit plus de produire des logiciels indépendants, mais de réaliser des compositions cohérentes de

systèmes pré-existants [Botterweck 13] et d'intégrer éventuellement de multiples LPL distinctes [Ferreira Filho 12].

Nous parlons ainsi dans nos travaux de *LPL complexe* pour désigner des LPL spécifiquement adaptées à la réalisation de systèmes-de-systèmes.

1.2 Défis et éléments de contribution

Les systèmes-de-systèmes étant par nature complexes, la réalisation et l'utilisation d'une LPL dédiée à ce type de système pose de nouveaux défis. A l'instar de Botterweck [Botterweck 13], nous caractérisons ces défis selon deux aspects : la modélisation de la variabilité et la configuration d'un produit.

1.2.1 Modélisation de la variabilité des systèmes-de-systèmes

Les systèmes-de-systèmes se définissent comme des assemblages de sous-systèmes dont chacun peut présenter de multiples variantes. Par ailleurs, la manière dont les sous-systèmes se combinent constitue également en soi une forme de variabilité. Ainsi, la représentation de la variabilité peut être très complexe, puisqu'elle est constituée par l'ensemble des combinaisons possibles de toutes les variantes du système-de-systèmes. Le problème est d'autant plus complexe que les différents éléments de variabilité sont généralement interdépendants.

Le premier défi de nos travaux est donc : *comment modéliser de manière cohérente la variabilité d'un système-de-systèmes au sein d'une LPL ?*

1.2.2 Configuration d'un système-de-systèmes

Cependant, la modélisation de cette variabilité ne sert pas uniquement de représentation abstraite à titre de documentation ou de communication. Elle doit aussi supporter un processus de configuration en garantissant la cohérence des produits à réaliser. Il s'agit dans le cadre d'une LPL d'utiliser la modélisation de la variabilité au travers d'un processus de configuration cohérent afin de réaliser des produits.

Or, comme l'explique Botterweck, les différents choix à réaliser sont interdépendants, toutes les combinaisons n'étant pas valides. La cohérence des configurations est alors d'autant plus importante que la complexité des systèmes exige une détection au plus tôt dans le processus de configuration de la "réalisabilité" des produits. En effet, face à la complexité des variantes qui composent un système-de-systèmes, il n'est pas pertinent d'attendre le déploiement pour découvrir les problèmes. Ainsi la complexité des systèmes-de-systèmes doit être appréhendée dans la ligne elle-même et ne peut pas être seulement déléguée à un processus de réalisation tardif.

De plus, le processus de configuration en lui-même peut être problématique. En effet, définir un workflow de configuration dépend des acteurs ciblés et dans les cadre des systèmes-de-systèmes, il est nécessaire que de nombreuses variantes du workflow de configuration lui-même soient proposées. En outre, il est probable que de nombreux acteurs soient impliqués dans la réalisation d'un système-de-systèmes et que des choix doivent être faits en parallèles à plusieurs niveaux différents en même temps. La flexibilité du processus de configuration est donc essentielle dans le cadre des systèmes-de-systèmes.

Notre second défi est donc : *comment permettre un processus de configuration qui soit à la fois cohérent, flexible et utilisable dans le cadre d'une LPL complexe ?*

1.2.3 Introduction de la contribution

Nous proposons dans cette thèse de définir une nouvelle approche permettant la modélisation d'une LPL complexe par la définition d'un modèle du domaine et l'utilisation de plusieurs modèles de variabilités liés entre eux. En particulier nous définissons des mécanismes pour garantir la cohérence de la LPL ainsi modélisée. Nous définissons également un processus de configuration garantissant la cohérence des choix en permettant la prise de décision sans ordre imposé et l'annulation des choix à n'importe quelle étape du processus.

Ce travail a été formalisé, mis en œuvre puis appliqué à une LPL de systèmes de diffusion d'informations présentée dans la partie de validation de ce manuscrit.

1.3 Exemple fil rouge

Nous nous appuierons dans l'ensemble de ce document sur un exemple fil rouge afin d'illustrer nos travaux.

De nombreux travaux dans le cadre des LPL prennent l'exemple domotique d'une *SmartHome*, une "maison intelligente", afin d'illustrer leurs approches sur la gestion de la variabilité [Pohl 05, Classen 08, Arboleda 13]. Dans le but d'illustrer une approche de LPL dédiée à des systèmes-de-systèmes, nous utiliserons un exemple similaire à celui de Possompès *et al.* sur les *SmartBuildings* [Possompès 13].

Nous souhaitons ainsi réaliser une LPL pour la modélisation de bâtiments contenant des appartements dits "intelligents" : c'est-à-dire des appartements dotés de capteurs et d'actuateurs pour adapter automatiquement la luminosité, la température ou déclencher des alarmes. Nous affirmons par ailleurs que les besoins des appartements et de l'immeuble peuvent être intrinsèquement liés : le chauffage peut, par exemple, être géré globalement pour tout l'immeuble ou localement pour chacun des appartements.

Cette LPL reste cependant une illustration et sera à ce titre limitée en nombre de caractéristiques. Cependant, la très grande variabilité des capteurs et actuateurs existants et les très nombreux scénarios que l'on peut envisager dans le cadre domotique en fait un exemple plausible de cas d'utilisation industrielle. Par ailleurs, les choix que nous faisons de variabilité et de relations entre les différents concepts que nous allons définir ont tous un but pédagogique pour illustrer nos différentes contributions.

1.4 Structure de la thèse

Cette thèse se compose de trois parties.

La première partie (Partie I) décrit l'état de l'art de cette thèse. Elle est constituée des trois chapitres suivants :

- chapitre 2 : Nous présentons dans ce chapitre l'état de l'art relatif à la modélisation de la variabilité dans les LPL.
- chapitre 3 : Nous présentons dans ce chapitre l'état de l'art relatif à la gestion du processus de configuration dans les LPL.
- chapitre 4 : Nous présentons dans ce chapitre le positionnement de nos travaux relativement à l'état de l'art, en définissant et en synthétisant nos objectifs.

La seconde partie (Partie II) présente les travaux de contribution de cette thèse. Elle est constituée des trois chapitres suivants :

— chapitre 5 : Nous présentons dans ce chapitre nos contributions relativement à la modélisation de la variabilité dans les LPL complexes.
— chapitre 6 : Nous présentons dans ce chapitre nos contributions relativement au processus de configuration dans les LPL complexes.
— chapitre 7 : Nous présentons dans ce chapitre une implémentation de nos différentes contributions, ainsi que des éléments additionnels répondant à nos objectifs.

Finalement la troisième partie (Partie III) présente la validation de nos contributions de thèse relativement aux objectifs définis dans notre chapitre de positionnement (chapitre 4) et appliqués à une LPL de portée industrielle appelée YOURCAST. Elle est constituée des deux chapitres suivants :

— chapitre 8 : Nous présentons dans ce chapitre la modélisation de la LPL YOURCAST ainsi que les résultats obtenus.
— chapitre 9 : Nous présentons dans ce chapitre les résultats obtenus lors de différentes configurations réalisées dans le cadre de la LPL YOURCAST.

Première partie

État de l’art

CHAPITRE 2

REPRÉSENTATION DE LA VARIABILITÉ DANS LES LPL

Comment modéliser de manière cohérente la variabilité d'un système-de-systèmes au sein d'une LPL ?

Sommaire

Résumé Nous réalisons dans ce chapitre un état de l'art de la modélisation de la variabilité dans les LPL. Nous revenons ainsi sur la définition même des LPL et sur leur principe de fonctionnement avant de nous intéresser plus particulièrement aux différents travaux concernant la modélisation de la variabilité.

Nous dégageons ainsi, à partir des différents travaux existants, les hypothèses de travail qui nous permettront de définir les objectifs de nos propres travaux pour répondre aux défis que nous avons explicités en introduction.

2.1 Introduction

Nous présentons dans ce chapitre un état de l'art de la représentation de la variabilité dans les LPL, en nous attachant à mettre en évidence nos hypothèses de travail concernant la définition des LPL complexes.

Nous avons défini en introduction le terme "LPL complexes" par des lignes de produits logiciels visant à réaliser des systèmes par composition de systèmes existants, et que nous pouvons qualifier de *systèmes-de-systèmes*.

Nous proposons dans ce chapitre un retour sur les lignes de produits logiciels en les définissant et en expliquant brièvement leur principe de fonctionnement. Nous donnons ensuite notre définition de la variabilité logicielle avant de réaliser un état de l'art sur les "*feature models*", qui sont parmi les modèles de variabilité les plus étudiés et utilisés. Nous discutons dans un troisième temps différents formalismes permettant de décrire des modèles de variabilité plus complexes et nous présentons leurs limitations. Enfin nous décrivons les travaux effectués permettant de décrire la variabilité dans les lignes de produits logiciels multiples.

2.2 Retour sur les LPL

Nous revenons dans cette section sur la définition des Lignes de Produits Logiciels (LPL) et leur principe général de fonctionnement.

2.2.1 Définition

Derrière l'expression *Ligne de Produits Logiciels* se cache un paradigme de production emprunté au taylorisme et appliqué à l'ingénierie du logiciel. L'idée derrière les LPL est de maximiser la production de logiciels similaires, issus d'une même famille de produits, par la réutilisation systématique d'une même base de code et par la définition de variantes.

On trouve dans la littérature plusieurs définitions permettant de qualifier les LPL. La définition de Clements et Northrop est très souvent citée dans les travaux du domaine :

> "*A* software product line *is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*
>
> [CLEMENTS 02]

Cette définition insiste sur le fait qu'un ensemble de systèmes à forte composante logicielle est développé à partir d'une base commune d'*assets*, ou artefacts logiciels. Les auteurs précisent également dans cette définition deux aspects organisationnels des LPL : celles-ci doivent être réalisées pour répondre à un secteur de marché précis et le développement d'un produit doit se faire de manière prescriptive, pour un besoin spécifiquement identifié.

Pohl *et al.* définissent quant à eux l'ingénierie des LPL comme un paradigme de développement de logiciels utilisant une plateforme logicielle commune, c'est-à-dire un ensemble de technologies sur lesquelles bâtir, et la personnalisation de masse [Pohl 05]. Ils réutilisent ainsi la définition de la personnalisation de masse proposée par Davis :

> “ *Mass customisation is the large-scale production of goods tailored to individual customers' needs.*
>
> [DAVIS 87]

La plupart des travaux sur les LPL font ainsi références à deux éléments clés pour la représentation d'une famille de logiciels : *(i)* un ensemble d'artefacts logiciels communs ; *(ii)* la possibilité de dériver un logiciel personnalisé, sur mesure, afin de répondre à des besoins spécifiques.

La littérature distingue ces deux éléments comme deux tâches distinctes : l'*ingénierie du domaine* et l'*ingénierie de l'application* [Pohl 05, Clements 02, Rashid 11, Arboleda 13].

HYPOTHÈSE DE TRAVAIL 1 :
CADRE DE TRAVAIL DES LPL DE SYSTÈMES-DE-SYSTÈMES

Nous étudions dans nos travaux les tâches liées à l'ingénierie du domaine et à l'ingénierie de l'application dans le cadre des LPL complexes. Nous considérons que les *produits* d'une LPL complexe sont des systèmes-de-systèmes. Par ailleurs, nous nous concentrons uniquement sur la formalisation des aspects techniques liés à la réalisation d'une telle LPL.

2.2.2 Ingénierie du domaine

L'*ingénierie du domaine* vise à définir précisément la famille de logiciels représentée par la Ligne de Produits Logiciels (LPL).

Arango discute en 1989 de la nécessité de formaliser l'ingénierie du domaine afin d'obtenir des informations explicites sur la réalisation d'un logiciel satisfaisant aux problèmes du domaine dans l'optique de favoriser la réutilisation [Arango 89]. Il propose ainsi un processus composé de trois activités :

— *l'analyse du domaine*, qui vise à réaliser un *modèle du domaine* comprenant des informations permettant à la fois de spécifier le système et de l'implémenter à partir de cette spécification ;
— la *spécification de l'infrastructure*, qui décrit l'organisation des éléments réutilisables ;
— l'*implémentation de l'infrastructure*, qui comprend l'implémentation des différents éléments réutilisables définis auparavant.

Arango discute dans cet article de la nécessité d'avoir une approche pratique pour l'analyse et la définition des informations du domaine [Arango 89]. En particulier, il insiste sur la difficulté à parvenir à une définition consensuelle concernant cette activité : il propose donc de s'attacher à ne représenter que les informations utiles pour satisfaire un but précis. Il sépare ainsi nettement les conceptions “pures” et “pratiques” de l'analyse du domaine. Nous conserverons dans l'ensemble du document cette vision d'Arango que nous qualifierons de *pragmatique*.

Une autre activité inhérente à l'ingénierie du domaine consiste à déterminer les limites de la famille de produits : quels sont les produits à inclure et ceux à ne pas prendre en compte ? Cette activité est appelée le *scoping* dans la littérature [John 09]. Celui-ci est déterminant à la

fois pour réaliser correctement l'ingénierie du domaine mais également pour assurer la viabilité économique de la ligne de produits : un scoping trop large aboutirait à inclure des produits dans la famille, sur lesquels le retour sur investissement serait extrêmement faible ; à l'inverse un scoping trop restreint oublierait certains produits phares qu'il serait alors nécessaire de développer manuellement, en perdant les bénéfices apportés par la ligne [Clements 02].

Nous pouvons ainsi redéfinir l'ingénierie du domaine dans le cadre des LPL sous la forme de 3 activités différentes de celles exposées par Arango :

1. le scoping permet en premier lieu de déterminer les limites de la famille de produits et influence les décisions suivantes pour la définition de la LPL ;
2. la définition d'une infrastructure de code commune à tous les produits ainsi que des différentes variantes de code permet d'automatiser la réutilisation ;
3. la modélisation de la variabilité de la ligne de produits et des relations entre les différents éléments de variabilité selon une représentation abstraite permet d'exploiter cette variabilité.

HYPOTHÈSE DE TRAVAIL 2 :
MODÉLISATION PRAGMATIQUE DE LA LPL

Dans le contexte de cette thèse nous nous concentrons uniquement sur l'activité de modélisation de la variabilité : c'est en effet cette étape qui constitue le premier défi présenté dans l'introduction auquel nous souhaitons répondre. Par ailleurs, dans la lignée des travaux d'Arango nous souhaitons réaliser une approche pragmatique de l'ingénierie du domaine dans le cas d'une LPL complexe. Enfin, la modélisation de la LPL sera effectuée par une personne que l'on appelera "*l'architecte de la LPL*".

2.2.3 Ingénierie de l'Application

L'*ingénierie de l'application* vise à permettre la réalisation de produits appartenant à la famille de produits décrite lors de l'ingénierie du domaine. La réalisation du produit durant l'ingénierie de l'application est généralement automatisée à partir des choix effectués au sein de la représentation abstraite définie lors de l'ingénierie du domaine [Rashid 11, Rabiser 10a].

Nous retrouvons dans la littérature deux activités essentielles liées à l'ingénierie de l'application :

— la sélection des fonctionnalités du produit à réaliser, aussi appelée *processus de configuration* et
— la réalisation finale du produit à partir de cette configuration et des assets de code associés, phase que nous adresserons en parlant de *processus de réalisation*.

La réalisation d'un produit est généralement effectuée de manière automatisée à partir d'un ensemble de choix effectués lors du processus de configuration. Cette étape fait donc appel à de nombreuses techniques avancées de l'ingénierie logicielle, telles que l'ingénierie des modèles, la génération de code, la programmation par aspect, ou par composants, ou encore le chargement dynamique de code.

Par ailleurs, le vocabulaire concernant cette étape n'est pas complètement fixé dans la littérature : si certains travaux comme ceux d'Arboleda et Royer parlent de *dérivation* (voir la définition 2.10 dans [Arboleda 13]), d'autres travaux comme la synthèse d'état de l'art de

Rabiser *et al.* parlent de *dérivation* de produit pour qualifier à la fois la configuration et la réalisation : *"Product derivation is a key activity in application engineering and addresses the selection and customization of assets from the product line"* [Rabiser 09]. Dans le cadre de nos travaux, nous utiliserons le terme de *dérivation* dans la même acception que Rabiser *et al.* : la *dérivation* est ainsi constituée des processus de *configuration* et de *réalisation*.

L'essentiel des gains apportés par les LPL se retrouve ainsi durant la phase de l'ingénierie de l'application : c'est en effet durant cette phase qu'un produit concret est dérivé d'une famille de produits. Deelstra *et al.* comparent les bénéfices apportés par l'ingénierie de l'application par rapport à l'investissement requis pour créer la LPL :

> *"The idea behind this approach to product engineering is that the investments required to develop the reusable artifacts during domain engineering, are outweighed by the benefits in deriving the individual products during application engineering.*
>
> [DEELSTRA 05]

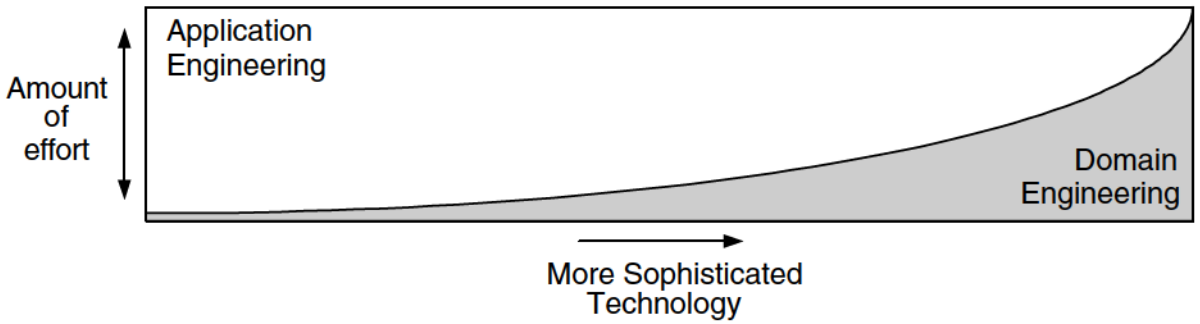


FIGURE 2.1 – Rapport entre les efforts fournis dans l'ingénierie du domaine et de l'application, issue de [Deelstra 05]

La Figure 2.1 issue des mêmes travaux montre que plus l'investissement est élevé dans l'ingénierie du domaine, moins l'ingénierie de l'application demandera d'efforts. Cependant, cette figure montre également que le gain est proportionnel à l'investissement technologique engagé dans la ligne de produits.

Enfin, l'étude réalisée en 2010 par Rabiser *et al.* montre que, alors que de nombreux travaux portent sur l'ingénierie du domaine, peu de travaux en comparaison traitent de la dérivation de produits [Rabiser 10a], alors même qu'il s'agit d'une activité essentielle des LPL.

HYPOTHÈSE DE TRAVAIL 3 :
FOCUS SUR LE PROCESSUS DE CONFIGURATION

Nous nous concentrons dans le cadre de nos travaux uniquement sur les problématiques liées à la configuration de produits et ne contribuons pas au processus de réalisation. Nous présentons cependant dans la validation une LPL complète intégrant la réalisation des produits. Par ailleurs, nous considérons que le processus de configuration est conduit par un "*utilisateur de la LPL*".

Nous revenons dans le chapitre suivant sur l'état de l'art lié à la configuration de produits.

2.3 Variabilité et Feature Models

Comme nous l'avons vu précédemment, l'ingénierie du domaine s'attache notamment à représenter les informations du domaine de la LPL et en particulier à modéliser sa variabilité. Nous revenons ici sur la définition de la variabilité logicielle avant de discuter différents formalismes de représentation de la variabilité.

2.3.1 Définition de la variabilité

La variabilité logicielle est définie par Jilles van Gurp *et al.* comme la capacité à changer ou personnaliser un système [van Gurp 01]. Pohl *et al.* offrent une définition détaillée de la variabilité en partant de trois questions [Pohl 05] :

— Qu'est-ce qui varie ?
— Pourquoi est-ce que cela varie ?
— Comment cela varie-t-il ?

Ils proposent ainsi de distinguer le "sujet de la variabilité" (ce qui varie), de "l'objet de la variabilité" (comment cela varie) et du "point de variation" (comment la variabilité est réalisée au sein de la ligne).

Arboleda *et al.* proposent une définition concise de la variabilité dans le cadre des LPL :

> *The variability of a set of software systems or products is the set of differences, described in a structured way, of some or all of their characteristics.*
>
> [ARBOLEDA 13]

Dans un article récent, Metzger et Pohl établissent une distinction intéressante entre la variabilité d'un logiciel et la variabilité d'une ligne de produits logiciels [Metzger 14]. Ainsi la variabilité d'un logiciel se réfère à sa capacité à être personnalisé, modifié, étendu ou configuré. Cette possibilité peut être offerte par des mécanismes bien connus de l'ingénierie logicielle comme l'utilisation d'interfaces, d'abstractions de classes, la compilation conditionnelle ou le chargement dynamique de classes. En revanche la variabilité au sein d'une LPL va permettre de représenter les différences et les éléments communs entre différents produits de cette LPL. Ainsi, si une intersection existe entre la variabilité d'un logiciel et la variabilité de sa LPL, certains éléments de variabilité logicielle peuvent ne pas être représentés comme une variabilité dans la LPL. On retrouve ici des problématiques liées au scoping de produits : l'infrastructure du logiciel peut avoir prévu des mécanismes de variabilité permettant de modéliser plus de produits que les produits qui ont été intégrés lors du scoping de la ligne. Nous nous attachons ici à représenter uniquement la variabilité des produits intégrés dans la LPL.

Par ailleurs, la gestion de la variabilité est une activité inhérente à l'utilisation de LPL. Pohl *et al.* la définissent comme l'ensemble des activités liées à la définition et l'exploitation de la variabilité au sein des LPL [Pohl 05]. On retrouve ainsi la distinction entre d'une part la définition de la variabilité liée à l'ingénierie du domaine, et d'autre part l'exploitation de cette même variabilité, durant l'ingénierie de l'application.

HYPOTHÈSE DE TRAVAIL 4 :
VARIABILITÉ DES SYSTÈMES

Dans le cas d'une LPL complexe, de nombreux éléments de variabilité doivent être pris en compte pour chacun des sous-systèmes de manière indépendante.

Nous décrivons dans la suite la représentation de la variabilité effectuée durant l'ingénierie du domaine.

2.3.2 FODA : les origines des FM

Les *Feature Models* [1] (FM) sont devenus un standard *de facto* pour représenter la variabilité.

Une très grande littérature a été réalisée autour de ce type de modèles et nous ne présenterons ici que les travaux les plus pertinents relativement à ce travail de thèse. Nous présentons donc dans la suite le formalisme à l'origine des FM, FODA, puis nous faisons un retour sur les formalismes étendus afin de permettre l'ajout de références et de cardinalités.

Kang *et al.* présentent en 1990 un rapport technique décrivant une approche d'analyse du domaine orientée features [2] (*Feature-Oriented Domain Analysis*, ou FODA) [Kang 90]. Ce document pose les bases du formalisme des FM.

Dans FODA, les features sont des propriétés d'un système visibles par l'utilisateur final :

> « *Feature : a prominent and user-visible aspect, quality, or characteristic of a software system or systems.*
>
> [KANG 90]

Un FM est défini par un *diagramme de features* et un *ensemble de contraintes*. Un diagramme de features est un arbre proposant une hiérarchie de features et dont la racine représente le concept que l'on cherche à modéliser. Des propriétés peuvent être précisées sur les features pour indiquer si elles représentent des caractéristiques optionnelles ou obligatoires, ou sur les groupes de features pour indiquer qu'une seule feature est autorisée parmi plusieurs : on parle de *ou-exclusif* ou XOR.

La relation hiérarchique permet de décomposer en niveaux d'abstraction : les nœuds de plus haut niveau représentent les caractéristiques de manière abstraite, alors que les feuilles représentent des caractéristiques plus concrètes. On retrouve ainsi représenté sous la forme d'un arbre à la fois les éléments communs à une famille de logiciels, représentés par des features obligatoires, et les éléments variants représentés par les features optionnelles et par les groupes XOR.

1. Le terme "feature model" est souvent traduit *modèle de caractéristique* ou *modèle de variabilité* en français. La deuxième expression nous semble trop large (d'autres types de modèles de variabilité existent !) et la première ne nous semble pas facilement représentative. Nous conserverons donc dans l'ensemble du document le terme *Feature Model* et l'acronyme FM qui lui correspond.

2. Le mot "feature" signifiant aussi bien les termes *fonctionnalité* que *caractéristique* en français, nous parlerons de *feature* dans l'ensemble du document pour conserver cette polysémie (ce qui est également justifié par la conservation de l'expression *Feature Model*).

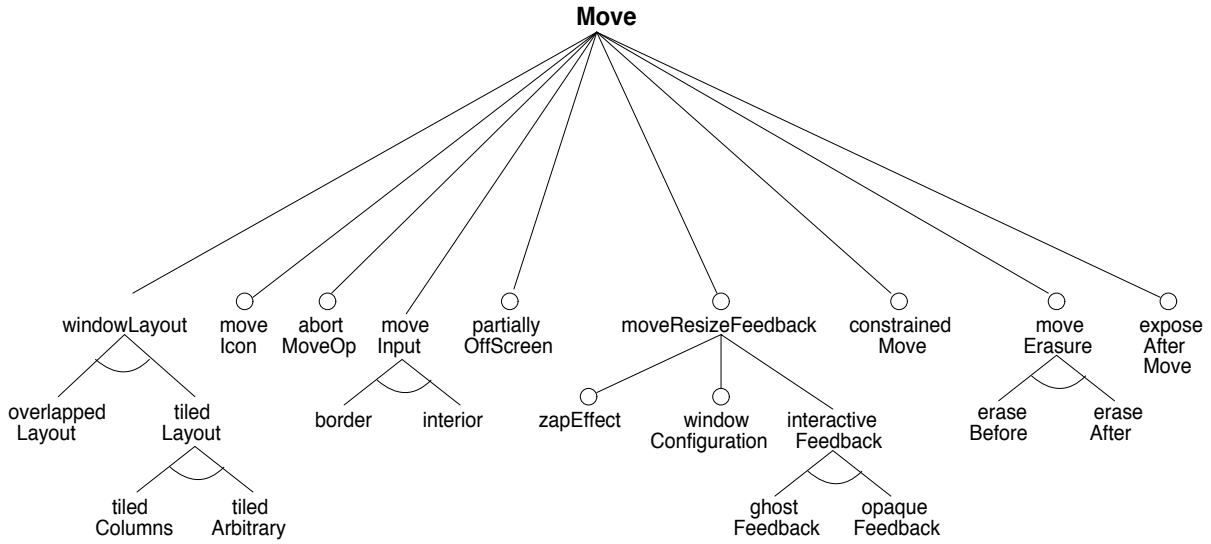


FIGURE 2.2 – Diagramme de feature extrait de [Kang 90]

La Figure 2.2 montre une représentation graphique d'un diagramme de features dans FODA, utilisé pour représenter l'opération de déplacement d'une fenêtre (*Move*) dans un gestionnaire de fenêtre. Les features optionnelles sont représentées avec un cercle blanc (par exemple `move icon`), et les groupes XOR sont représentés avec un arc de cercle (par exemple entre `border` et `interior`). Les features sans formalisme graphique sont obligatoires (par exemple `move input`). Ce formalisme graphique est repris dans la plupart des travaux qui ont suivi, à l'exception de la notation pour les features obligatoires qui est ensuite généralement représentée avec une pastille noire.

Les contraintes ajoutées au diagramme de features peuvent exprimer deux choses dans le formalisme de Kang *et al.* : des exigences (*requires*) ou des exclusions (*mutex-with*) entre features. On retrouve dans [Kang 90] les exemples suivants de contraintes liés à la Figure 2.2 :

— "`opaqueFeedback mutex-with moveErasure`" : la première feature (*opaqueFeedback*) implique que le mouvement d'une fenêtre est visible lors de son déplacement par la superposition de l'image de la fenêtre le long du mouvement de la souris ; la seconde feature (*moveErasure*) indique que l'image de la fenêtre est supprimée lors du déplacement. Ces deux features sont donc contradictoires : la feature *moveErasure* ne peut pas être utilisée dans le cas d'un *opaqueFeedback* car on souhaite conserver les images de la fenêtre lors du mouvement.

— "`ghostFeedback requires moveErasure`" : à l'inverse un *ghostFeedback* implique que l'on peut suivre le mouvement d'une fenêtre lors du déplacement en ne conservant que quelques images : il est alors indispensable de pouvoir supprimer les images en trop grâce au *moveErasure*.

Un FM est ainsi déterminé à partir du diagramme de features représenté dans la Figure 2.2 et les contraintes discutées ci-dessus. Nous utilisons dans la suite du document uniquement le terme FM, même dans le cas où il n'existe pas de contraintes supplémentaires au diagramme de features.

Le but d'un FM est de fournir une représentation synthétique des features disponibles pour un concept et des relations qui existent entre les features. Ainsi, dans le cadre des LPL, le FM représente un ensemble de produits disponibles, chaque produit étant représenté par une combinaison *valide* de features. La validité de la combinaison de features dépend des contraintes exprimées soit dans les propriétés du diagramme de features, soit dans les contraintes

additionnelles du FM. On parle généralement de *configuration* [3] pour qualifier une combinaison de features d'un FM.

Schobbens *et al.* ont réalisé une analyse de la sémantique des différents diagrammes de features existants [Schobbens 07]. On constate qu'en plus des propriétés définies dans FODA, la plupart des formalismes intègrent la notion de groupe de features autorisant une ou plusieurs features : on parle alors de groupe OR. Ainsi de nombreux formalismes de FM existent [Batory 05, Acher 11, Czarnecki 05a, Classen 11].

Enfin, Benavides *et al.* ont réalisé un état de l'art sur les travaux d'analyse automatique des FM [Benavides 10]. Ils définissent dans cet article les différentes opérations d'analyses possibles sur un FM, notamment afin de permettre la détection d'erreurs telles que les features mortes (features absentes de toute configuration valide), les faux optionnels (features optionnelles dans la hiérarchie mais obligatoires par les contraintes) ou encore les FM ne permettant aucune configuration valide. Les auteurs mettent en avant le fait qu'analyser un FM manuellement est une activité fastidieuse et source d'erreurs : il est en effet difficile de prendre en compte les nombreuses combinaisons de features à vérifier en s'assurant de la cohérence des contraintes.

HYPOTHÈSE DE TRAVAIL 5 :
COHÉRENCE D'UNE LPL COMPLEXE

Nous nous intéressons ainsi dans nos travaux à vérifier de manière automatique la représentation de la variabilité d'une LPL complexe afin de s'assurer de sa cohérence, au sens où l'architecte de la LPL doit pouvoir vérifier que toute configuration valide est atteignable.

2.3.3 Cardinalités, Attributs et Références

Largement adopté pour la modélisation de la variabilité dans les LPL, FODA montre cependant rapidement ses limites. En effet, le formalisme ne permet pas d'exprimer des multiplicités [4] sur les features afin de signifier qu'une caractéristique peut être autorisée plusieurs fois au sein d'une configuration.

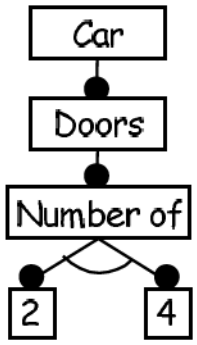


FIGURE 2.3 – Exemple d'un FM possédant des cardinalités représentées comme des features. Extrait de [Czarnecki 02]

En 2000, Czarnecki suggère d'utiliser des features pour modéliser des cardinalités dans les FM [Czarnecki 00]. Par exemple, la Figure 2.3 décrit une voiture possédant 2 ou 4 portes. Cependant, cette modélisation est très peu expressive comme le souligne Czarnecki dans [Czarnecki 02] : elle ne fonctionne pas si la feature dont on souhaite exprimer la cardinalité est racine d'un sous-arbre présentant des choix. En outre, elle limite l'expression à des cardinalités fixées ce qui empêche la définition de cardinalités sous forme d'intervalles.

3. Nous distinguons dans ce document le *processus de configuration* qui consiste à effectuer des choix pour la réalisation d'un produit, et la *configuration* qui est le résultat de ce processus.

4. Le problème concerne ici la modélisation des *multiplicités* au sein des FM, afin de s'assurer que la *cardinalité* des features dans les configurations respectent ces multiplicités. Nous utilisons indifféremment les termes *cardinalité* et *multiplicité* dans la suite afin d'être cohérent par rapport aux travaux du domaine que nous citons.

Riebisch *et al.* proposent en 2002 d'étendre la sémantique des diagrammes de features pour intégrer des multiplicités UML [Riebisch 02]. Ils souhaitent en premier lieu rendre plus explicite la sémantique des groupes de features : par exemple, un groupe XOR signifie une multiplicité `1..1`. Ils souhaitent également permettre des multiplicités intermédiaires : par exemple d'exiger au moins 2 éléments dans un groupe OR de 4 features. Enfin, l'objectif de Riebisch *et al.* est d'unifier la notation des multiplicités dans les diagrammes de features, avec la notation standardisée employée en UML.

Suite à leur article de 2002 motivant le besoin de cardinalités [Czarnecki 02], Czarnecki *et al.* proposent en 2004 un formalisme de FM étendu permettant non seulement de spécifier la multiplicité sur des nœuds représentant des groupes de features, mais également sur des nœuds représentant une hiérarchie unique ou sur des feuilles. Il est ainsi possible de représenter des familles de produits plus complexes grâce aux multiplicités.

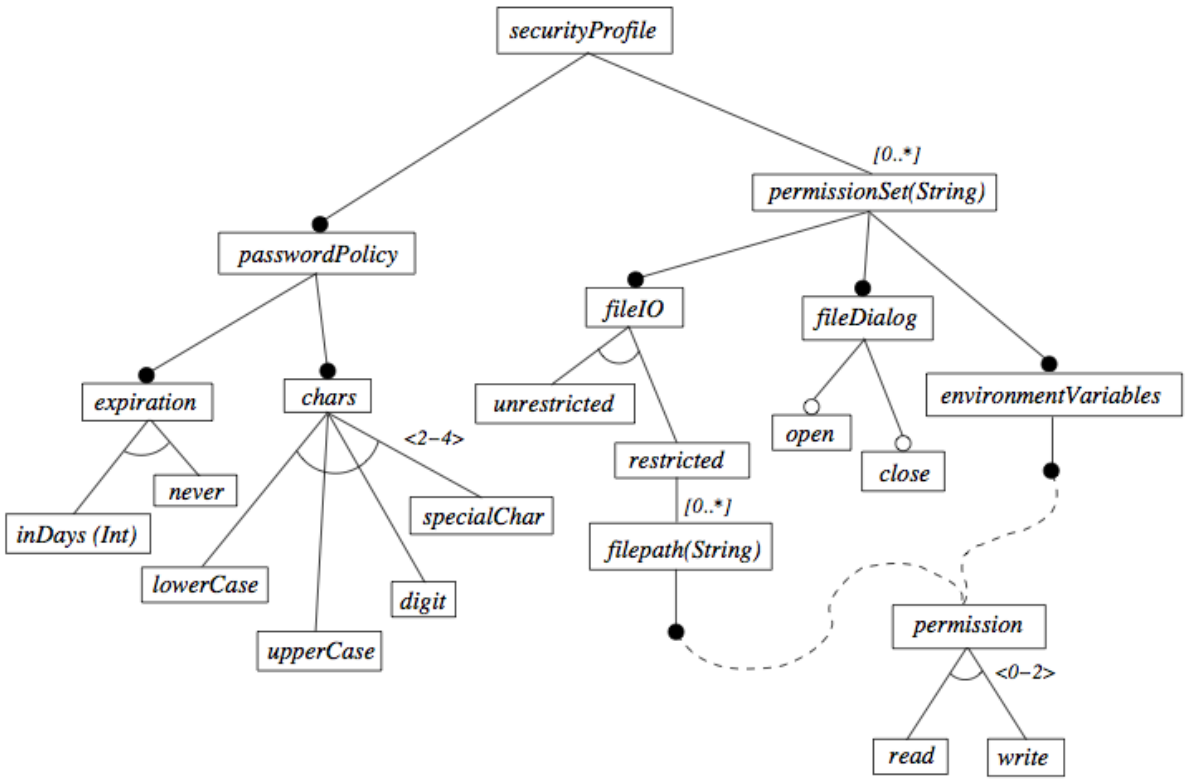


FIGURE 2.4 – Exemple d'un FM de profil de sécurité extrait de [Czarnecki 05b]

Par exemple, la Figure 2.4 représente une famille de produits permettant de gérer des profils de sécurité. La cardinalité spécifiée sur la feature *permissionSet* indique que cette feature pourra être intégrée un nombre illimité de fois à un produit *avec toutes les caractéristiques qu'elle contient* : c'est-à-dire, le fait qu'elle supporte une boîte de dialogue de fichier (*fileDialog*), une entrée/sortie sur les fichiers (*fileIO*), etc.

On peut cependant évoquer les travaux de Michel *et al.* à propos des cardinalités dans les FM, qui montrent que leur interprétation peut être ambigüe [Michel 11].

La Figure 2.5 montre ainsi un FM très simple dont la composition des cardinalités est ambigüe : doit-on considérer les cardinalités de C relativement au FM complet ou relativement à la feature B ? Ainsi, si les cardinalités dans les FM offrent davantage d'expressivité, leur utilisation reste sujette à interprétation.

Par ailleurs, Czarnecki *et al.* expliquent dans leurs travaux l'utilité de pouvoir ajouter des

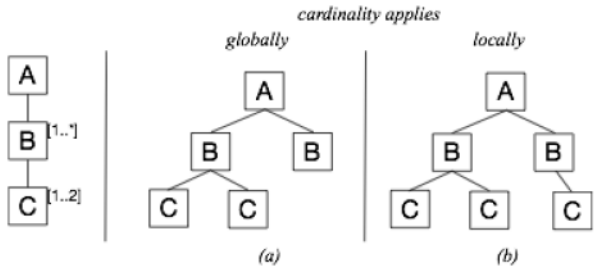


FIGURE 2.5 – Illustration de l'ambiguité des cardinalité extraite de [Michel 11]

attributs aux features afin de pouvoir leur donner une valeur : la variabilité n'est plus seulement contenue dans le FM mais elle est également représentée par l'ensemble des valeurs acceptées pour les différentes features à valuer [Czarnecki 05b]. Dans l'exemple de la Figure 2.4 cela signifie que le *permissionSet* contient comme élément de variabilité une chaîne de caractères. Dans ce même exemple on voit que la feature `inDays` spécifie comme élément de variabilité un entier.

Enfin, les auteurs présentent un mécanisme de modularisation des FM en introduisant la possibilité de réaliser des références entre les FM : une feature peut faire référence à un FM externe, afin d'éviter la redondance d'information dans le FM et de faciliter la réutilisation. Dans la Figure 2.4 les features `filepath` et `environmentVariables` font ainsi toutes deux références à un FM `Permission`.

> *This mechanism allows breaking up large diagrams into smaller ones and reusing common parts in several places. This is an important mechanism because, in practice, feature diagrams often become too large to be considered in their entirety.*
>
> [CZARNECKI 05B]

Ainsi le mécanisme de référence permet de garantir une bonne séparation des préoccupations dans les FM. Il est possible de définir des FM de plus petite taille n'ayant que la préoccupation d'un domaine très spécifique et qui pourra être par la suite réutilisé dans différents FM.

HYPOTHÈSE DE TRAVAIL 6 :
MULTIPLICITÉ ET SÉPARATION DES PRÉOCCUPATIONS

Il nous semble important d'être en mesure de gérer des multiplicités sous forme d'intervalles entre les composants d'un système-de-systèmes. Par ailleurs, la séparation des préoccupations est un point qu'il est essentiel de prendre en compte afin de gérer la variabilité d'un système très complexe. Nous considérons ces deux points dans nos travaux.

2.4 Autres approches de modélisation de la variabilité

Bien que le formalisme des FM soit le modèle de variabilité le plus discuté dans la littérature et le plus utilisé, d'autres formes de modélisation de la variabilité ont été définies afin de répondre à différentes problématiques, comme le montrent Berger *et al.* dans une étude basée sur les pratiques industrielles [Berger 13].

2.4.1 Modèles de décision

Certains travaux exploitent des *modèles de décision* afin de représenter la variabilité d'une LPL. On peut en particulier citer les travaux de Dhungana *et al.* concernant la solution DOPLER [Dhungana 10].

Historiquement les modèles de décision dérivent tous de la méthode Synthesis développée par le Software Productivity Consortium [Campbell 90]. On retrouve dans cette méthode la définition suivante pour qualifier les modèles de décision :

> *A set of decisions that are adequate to distinguish among the members of an application engineering product family and to guide adaptation of application engineering work product.*
>
> [CAMPBELL 90]

Ainsi les modèles de décision ne sont pas définis relativement à l'ensemble des produits à représenter, mais sont définis en fonction des décisions qui devront être prises lors du processus de configuration.

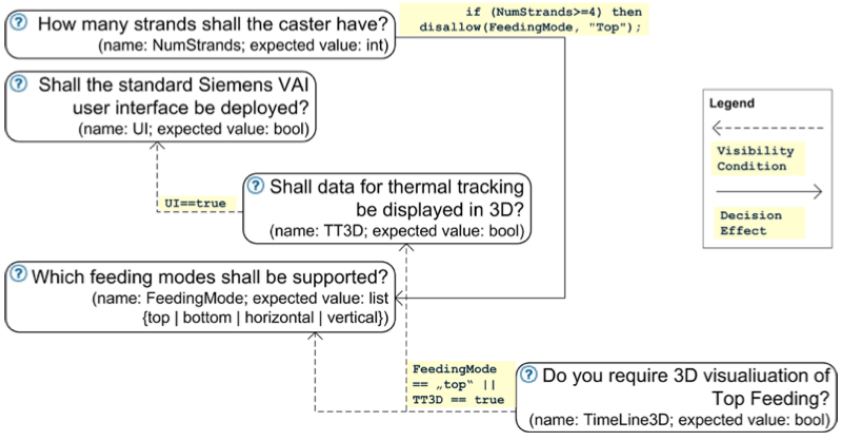


FIGURE 2.6 – Illustration d'un modèle de décision extrait de [Dhungana 10]

La Figure 2.6 présente ainsi un modèle de décision utilisé pour une ligne de produit de Siemens VAI. On constate que les décisions sont formulées sous forme de questions dont les réponses peuvent être des valeurs booléennes ou non. Par ailleurs les différentes décisions prises influencent à la fois la visibilité des futures questions, ainsi que les valeurs possibles. Les décisions sont donc organisées de manière hiérarchique mais celle-ci dépend des valeurs données aux décisions de plus haut niveau.

Cependant, malgré ces différences Czarnecki *et al.* montrent que les modèles de décision et les FM présentent des propriétés très similaires [Czarnecki 12].

HYPOTHÈSE DE TRAVAIL 7 :
POINT DE VUE DE L'UTILISATEUR FINAL SUR LA LPL

La modélisation de la variabilité d'un système-de-systèmes fait appel à de nombreux domaines d'expertises. Plutôt que de modéliser cette variabilité selon les domaines d'expertise, il nous semble plus pertinent dans nos travaux de modéliser la variabilité en cohérence avec les besoins et les exigences qu'un utilisateur final de la LPL va vouloir exprimer pour produire son système-de-systèmes.

2.4.2 Représentation orthogonale et unifiée de la variabilité

D'autres travaux, en revanche, proposent de suivre une approche radicalement différente en représentant la variabilité de manière orthogonale à la modélisation d'une application. On peut par exemple évoquer le formalisme défini par Pohl *et al.* , OVM (Orthogonal Variability Modeling) dont le but est justement de modéliser des points de variation qui conviennent aussi bien pour des modèles d'exigences, des modèles d'architecture de l'application, que pour des choix d'implémentations ou de tests [Pohl 05].

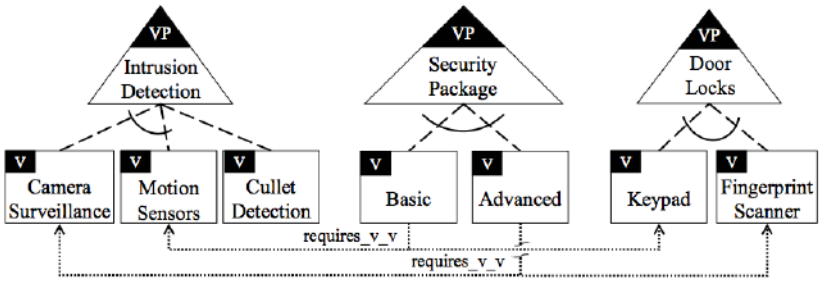


FIGURE 2.7 – Illustration d'un modèle OVM extrait de [Pohl 05]

La Figure 2.7 montre ainsi un exemple simple de modèle exploitant le formalisme OVM. Ce modèle comporte trois points de variation pour la détection d'intrusion, la gestion de la sécurité et le verrouillage des portes. Pour chacun de ces points de variation, le modèle propose deux ou trois variantes distinctes : par exemple, un scanner à empreinte digitale ou un pavé numérique pour la gestion du verrouillage des portes. Par ailleurs, des contraintes sont spécifiées entre les différentes variantes : le fait de prendre une gestion de la sécurité "basique" implique la variante de gestion du mouvement pour la détection d'intrusion et le pavé numérique pour le verrouillage.

En outre, on trouve de nombreux travaux autour de la représentation de la variabilité au sein même des modèles. Le langage CVL (Common Variability Language) proposé à la standardisation se veut ainsi être un DSL d'expression de la variabilité dans les modèles [OMG Revised Submission 12]. D'autres approches visent également à définir la variabilité en exploitant les profils UML [Possompès 11] ou encore la modélisation par aspect [Morin 09].

Dans une approche différente, Bak *et al.* proposent d'unifier la métamodélisation et la modélisation de la variabilité au sein d'une même sémantique définie par le formalisme CLAFER [Bąk 11]. Cette approche est intéressante en ce sens qu'elle permet de conserver une vision abstraite de haut niveau du domaine métier à modéliser, tout en permettant une modélisation de la variabilité très fine au niveau de la feature. Cependant, en confondant les deux approches au sein d'un même langage, Bak *et al.* réalisent un formalisme très puissant dont les concepts sont difficiles à appréhender et à exploiter, particulièrement dans le cas de familles de produits complexes impliquant de nombreuses features.

Enfin, on peut également évoquer les travaux exploitant des ontologies [Asikainen 07] ou des treillis de concepts [Al-msie'deen 14] afin de représenter la variabilité.

HYPOTHÈSE DE TRAVAIL 8 :
STANDARDS ET SÉPARATION DES FORMALISMES

L'adoption d'une approche repose grandement sur sa facilité d'utilisation. Ainsi l'exploitation de standards déjà existants et la conservation d'une séparation des formalismes dans la modélisation de la variabilité sont des pratiques que nous souhaitons conserver dans nos travaux.

2.5 Composition et Lignes de Produits Multiples

En 2002, Jan Bosch décrit six formes différentes de LPL, présentées dans la Figure 2.8, qu'il classe selon leur maturité [Bosch 02]. Il évoque ainsi les concepts d'*infrastructure standardisée*, correspondant à la standardisation des technologies employées dans les produits, et de *plateforme de produits*, que l'on pourrait comparer à un framework, comme les précurseurs de la LPL proprement dite.

Il parle en outre de *base de produits configurables* comme d'une LPL dans laquelle le domaine est parfaitement maîtrisée : tout l'effort est mis dans l'ingénierie du domaine, l'ingénierie de l'application est alors complètement automatisée. Cela correspond au discours de Deelstra concernant l'investissement dans la LPL, présenté dans la Figure 2.1 [Deelstra 05].

Nous discutons dans la suite des deux autres formes de LPL présentées par Bosch : les *populations de produits* et les *lignes de produits multiples* en postulant qu'il s'agit de deux formes différentes de LPL qu'il peut être intéressant d'exploiter afin de gérer la grande variabilité présente dans les systèmes-de-systèmes.

2.5.1 Populations de produits et Composition

La *population de produits* représente pour Bosch une forme possible de LPL. Il s'agit en fait d'étendre le scoping d'une LPL pour permettre de dériver d'autres types de produits en combinant différemment les artefacts logiciels déjà existants dans la ligne.

Ommering fait référence à cette problématique dans un article portant sur les produits électroménagers de divertissements réalisés par Philips embarquant du logiciel, comme les télévisions ou les magnétoscopes [van Ommering 00]. Ommering parle alors de sous-systèmes comme des composants logiciels complexes qui implémentent les fonctionnalités d'un sous-domaine. Un produit contient alors plusieurs de ces sous-systèmes.

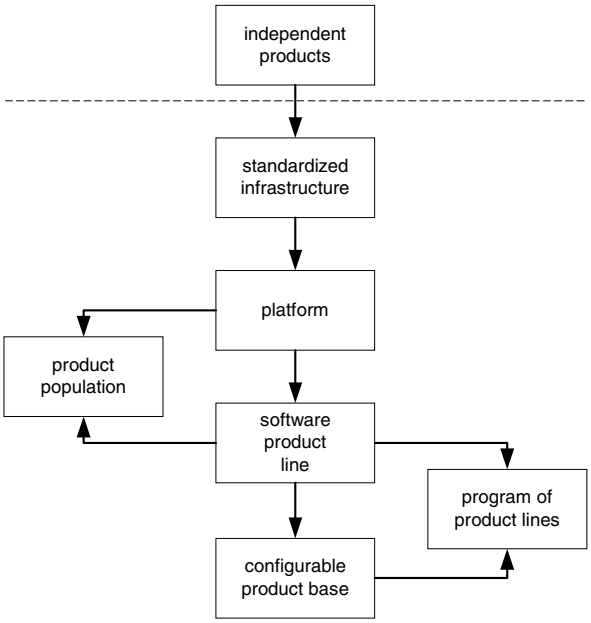


FIGURE 2.8 – Evolution des LPL. Extrait de [Bosch 02]

Une telle population de produits présente donc de la variabilité à deux niveaux : à la fois à l'intérieur des sous-systèmes, mais également dans la manière dont les sous-systèmes sont composés. Dans un article commun à Ommering et Bosch publié en 2002, la présence de la variabilité au niveau des compositions est explicitée :

> ***Composability is actually a form of variation !***
> *We have discussed variation and composition. We presented them as fundamentally different, but it is possible to unify the concepts. In all the examples that we have seen, components become 'freely composable' when context dependencies have been turned into explicit variation points.*
>
> [VAN OMMERING 02]

Dans une approche différente, Acher *et al.* proposent des outils permettant d'automatiser des opérations de composition sur les FM à travers un DSL dédié [Acher 12]. Ainsi il est possible de représenter automatiquement la variabilité d'une LPL à partir de la définition de l'ensemble des produits de cette LPL, par fusion des informations. Les FM n'ont alors plus à être définis de manière monolithique mais peuvent être librement composés, agrégés ou découpés selon les besoins de la LPL.

HYPOTHÈSE DE TRAVAIL 9 :
VARIABILITÉ DES COMPOSITIONS DE SYSTÈMES

Un système-de-systèmes est constitué par un assemblage de systèmes, chacun pouvant présenter des éléments de variabilité (voir Hypothèse 4). Nous souhaitons ainsi intégrer dans nos travaux la capacité à gérer la variabilité des fonctionnalités mais également des compositions de systèmes.

2.5.2 Lignes de Produits Logiciels Multiples

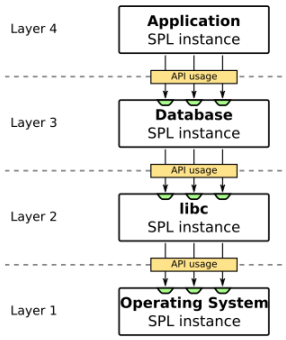
Bosch évoque dans ses différentes formes de LPL les *programmes de lignes de produits* [Bosch 02]. Il explique qu'il s'agit d'une approche dans laquelle l'architecture du logiciel est définie par une première LPL et que les différents composants eux-mêmes sont ensuites définis par d'autres LPL. Nous avons ainsi différentes LPL qui sont utilisées de manière conjointe et qui définissent des relations entre elles.

Cette approche est référencée dans de nombreux travaux sous le terme de *Lignes de Produits Logiciels Multiples* (LPLM). Höll *et al.* ont publié un l'état de l'art de ce type d'approche en 2012, et ils proposent la définition suivante pour les qualifier :

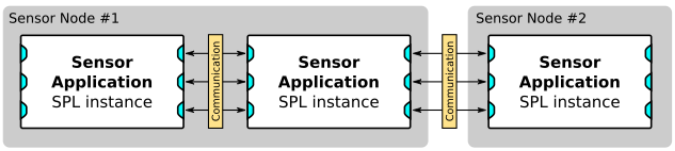
> “ *a set of several self-contained but still interdependent product lines that together represent a large-scale or ultra-large-scale system.*
>
> [HOLL 12]

De cet état de l'art, Höll *et al.* ont identifié une quinzaine d'approches différentes liées au lignes de produits multiples classées en fonction de leur problématique autour de l'ingénierie du domaine, de l'application ou sur des problématiques organisationnelles. La plupart des approches s'adressent à résoudre des problèmes liés à la taille et la complexité des systèmes [Hartmann 08, Acher 10a], au manque de gestion de dépendances entre les systèmes [Rosenmüller 08, Friess 07], au manque de modèles organisationnels [Jansen 09, Bosch 10], ou encore au manque de réutilisation et de possibilités de compositions dans les LPL [Dhungana 11, Leitner 11, Elsner 11, Schirmeier 09].

Dhungana *et al.* proposent une approche appelée *Invar* permettant de partager et réutiliser des modèles de variabilité à travers différentes LPL en exprimant des contraintes entre elles [Dhungana 11]. *Invar* permet de gérer un dépôt contenant à la fois des modèles de variabilité ainsi que des informations de dépendances entre ces modèles : les dépendances sont exprimées sous la forme “si ... alors ...”, différentes conditions et actions sur les modèles de variabilité étant supportées. L'approche pré-suppose que chaque modèle de variabilité dispose de son propre web service de configuration, un middleware de configuration est alors mis en place pour communiquer avec tous ces services tout en prenant en compte les informations de dépendances : une interface générique de configuration peut donc être mise en place pour cacher à l'utilisateur final toute la complexité de la LPL composée.



(a) Composition verticale de LPL



(b) Composition horizontale de LPL

FIGURE 2.9 – Compositions de LPL extraits de [Schirmeier 09]

Dans leurs travaux, Schirmeier et Spinczyk discutent des challenges posés par l'utilisation conjointes de plusieurs LPL [Schirmeier 09].

Ils comparent ainsi deux approches possibles des LPLM exploitant différentes LPL de manière verticale ou exploitant différentes instances d'un même type de LPL de manière horizontale (voir Figure 2.9). La première représente une LPL organisée comme une hiérarchie de différents produits de LPL, chaque produit d'une LPL ayant besoin des fonctionnalités apportées par le ou les produits du dessous. Ainsi les dépendances sont alors uniquement organisées de la LPL de plus haut niveau vers les LPL de plus bas niveau. La composition horizontale à l'inverse permet selon les auteurs de représenter des systèmes composés de plusieurs produits issus de la même LPL. Les auteurs précisent qu'il est alors nécessaire de définir à l'avance le nombre de produits d'une LPL qui seront utilisés dans le système final.

Leitner *et al.* proposent dans leurs travaux d'unifier différents paradigmes de modélisation afin de représenter les informations du domaine [Leitner 11]. En particulier ils montrent comment combiner un FM et un modèle spécifique du domaine afin de les utiliser en cohérence. Ils insistent dans leurs travaux sur leur volonté de ne pas inventer un nouveau langage, mais au contraire de réutiliser les outils déjà existants de modélisation de la variabilité.

Dans la continuité de ces travaux, Rosenmüller *et al.* proposent une approche permettant de définir un modèle objet spécifiant les différentes LPL à utiliser ainsi que les contraintes exprimées entre ces LPL [Rosenmüller 08].

Cette approche illustrée dans la Figure 2.10 permet de conserver à la fois une vision architecturale de haut niveau de la LPL en train d'être modélisée, tout en conservant des représentations standards de la variabilité. Par ailleurs, l'approche permet de spécifier différents niveaux de contraintes, aussi bien au niveau du domaine, qu'au niveau des différents composants représentés comme des LPL distincts. En revanche, le nombre de composants est figé dans le modèle d'architecture de la LPLM. Il est ainsi impossible de définir des multiplicités sous forme d'intervalles dans ce modèle d'architecture.

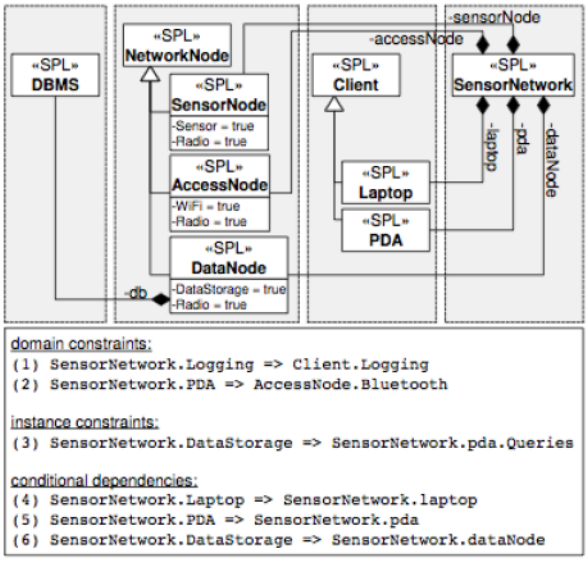


FIGURE 2.10 – Modélisation de LPLM par l'approche définie dans [Rosenmüller 08]

HYPOTHÈSE DE TRAVAIL 10 :
MULTIPLICITÉ NON BORNÉE ET SÉPARATION DES FORMALISMES

Nous ne considérons pas dans nos travaux l'utilisation d'une ligne de produits multiple, au sens où cette LPL serait créée par la combinaison de plusieurs LPL aussi bien du point de vue de la modélisation de sa variabilité, de la définition de son processus de configuration, qu'au niveau de la réalisation du produit. En revanche, les travaux effectués dans le cadre des LPLM sur la représentation de la variabilité par composition et la définition des contraintes entre plusieurs modèles de variabilité nous semblent particulièrement important pour garantir la séparation des préoccupations et des formalismes dans une LPL complexe. Nous souhaitons ainsi nous appuyer sur les travaux de Leitner *et al.* [Leitner 11] et Rosenmüller *et al.* [Rosenmüller 08] afin de supporter à la fois la séparation des préoccupations et des formalismes tout en réutilisant les outils déjà existants de modélisation de la variabilité.
En outre, il nous semble essentiel dans le cadre de la modélisation de la variabilité d'un système-de-systèmes de ne pas fixer la cardinalité des éléments, comme nous l'avons vu dans les travaux du domaine, mais de proposer une multiplicité sous la forme d'un intervalle.

2.6 Conclusion

Nous avons réalisé dans ce chapitre un état de l'art sur la modélisation de la variabilité des lignes de produits logiciels, en nous intéressant plus particulièrement à la problématique de modélisation des LPL complexes.

Nous avons ainsi défini nos hypothèses de travail en fonction des travaux déjà réalisés dans le domaine. Nous nous intéressons en particulier aux derniers travaux réalisés relatifs à la

définition de la variabilité à travers de multiples modèles, même si ceux-ci présentent encore des limitations que nous tâcherons de dépasser dans nos travaux.

CHAPITRE 3

PROCESSUS DE CONFIGURATION DES LPL

Comment permettre un processus de configuration cohérent, flexible et utilisable dans le cadre d'une LPL complexe ?

Sommaire

Résumé Nous poursuivons dans ce chapitre notre état de l'art en étudiant les travaux existants relatifs à la définition des processus de configuration. Nous revenons à cet effet sur la notion même de configuration afin de mieux l'appréhender.

Dans la continuité du chapitre précédent, nous mettons en évidence nos hypothèses de travail afin de positionner nos travaux.

3.1 Introduction

Le processus de configuration est une activité majeure de l'ingénierie de l'application qui consiste à choisir les fonctionnalités souhaitées au sein de la LPL avant la réalisation du produit. Nous avons présenté au chapitre précédent différents travaux permettant la modélisation de la variabilité au sein d'une LPL. Nous décrivons différents processus de configuration existants, en fonction des formalismes utilisés pour la réalisation de la LPL.

Nous revenons dans un premier temps sur la notion même de *configuration* qui, si elle est largement employée dans le domaine des LPL, a été également définie dans le domaine de l'intelligence artificielle. Nous décrivons ensuite les différents travaux traitant des mécanismes de raisonnement pour réaliser la configuration des FM, mais aussi les travaux autour de la définition même des workflows de configuration dans une LPL. Enfin nous revenons sur les mécanismes de configuration dans le cadre des lignes de produits multiples.

3.2 Retour sur la notion de configuration

Les notions de *configuration* et de *processus de configuration* ont été en premier lieu utilisées dans les domaines de l'intelligence artificielle et de la résolution de problèmes en recherche opérationnelle, principalement pour permettre la configuration de produits physiques. Ainsi en 1998, Faltings et Freuder introduisent un numéro spécial de *Intelligent Systems and their Applications* centré sur les processus de configuration [Faltings 98]. Les deux auteurs définissent ce processus de configuration comme un moyen de personnaliser des parties de produits afin de répondre aux besoins spécifiques des consommateurs. Ils insistent en particulier sur trois critères auxquels doit répondre une configuration : *(i)* elle doit être correcte afin que la compagnie puisse livrer le produit ; *(ii)* elle doit être produite rapidement, afin de ne pas perdre le client à la concurrence ; *(iii)* elle doit être optimale afin de convaincre le consommateur. Par ailleurs ils notent que ces critères favorisent grandement l'automatisation du processus de configuration et que les progrès du commerce électronique tendent à amplifier cette tendance.

Dans la même communauté d'intelligence artificielle, Ulrich Junker commence son chapitre sur la configuration dans *Handbook of Constraint Programming* avec cette définition :

> “ *Configuration is the task of composing a customized system out of generic components.*
>
> [JUNKER 06]

Dans un article récent de Hubaux *et al.* proposent d'unifier les travaux sur les configurations de produits, issus de l'intelligence artificielle, et ceux sur les configurations de logiciels, issus de l'ingénierie des lignes de produits logiciels [Hubaux 12a]. Ainsi Junker évoquait déjà en 2006 le problème de la sélection d'options dans des modèles booléens : il s'agit ici exactement du problème de la sélection des features dans un feature model [Junker 06]. Le processus de configuration au sein des LPL s'attache à réaliser un ensemble de choix dans le modèle de variabilité utilisé pour représenter la famille de produits. La configuration obtenue à partir de ce processus est ensuite utilisée pour le processus de réalisation du produit qui est généralement automatisé.

HYPOTHÈSE DE TRAVAIL 11 :
CONFIGURATION COHÉRENTE

Nous défendons dans notre approche le fait qu'une configuration doit être cohérente par rapport à la présentation de la variabilité dont elle est issue. Par ailleurs, nous considérons que toute configuration cohérente ne concerne qu'un produit atteignable de la famille de produits représentée.

3.3 Configuration de FM

Comme nous l'avons montré au chapitre précédent, la représentation de la variabilité dans les LPL passe essentiellement par l'utilisation de FM. Nous présentons ici les différents travaux traitant des mécanismes de raisonnement sur les FM et de la définition des processus de configuration pour les FM.

3.3.1 Sémantique formelle et raisonnement

Les FM sont utilisés pour modéliser la variabilité d'une LPL. Ils définissent ainsi un ensemble de produits réalisables au sein de la LPL. Choisir un produit dans cet ensemble revient alors à sélectionner des features exprimées dans le FM.

Dans leur état de l'art, sur les supports à la dérivation de produits, Rabiser *et al.* font la liste des fonctionnalités requises dans un outil de configuration [Rabiser 10a]. Le premier besoin qu'ils évoquent est celui de la résolution de la variabilité de manière automatique et interactive : l'utilisateur doit en premier lieu être en mesure de faire des choix, mais les outils doivent également résoudre les contraintes et dépendances afin d'indiquer à l'utilisateur la cohérence ou non de ses choix. Par ailleurs, les auteurs notent comme support avancé à ce besoin la capacité des systèmes de configuration à fournir un feedback immédiat à l'utilisateur. Enfin, un des besoin présenté dans cette étude concerne le guidage des utilisateurs durant le processus de configuration : les utilisateurs doivent en effet être aidés dans leurs choix en présentant les informations de variabilité de la manière la plus claire possible.

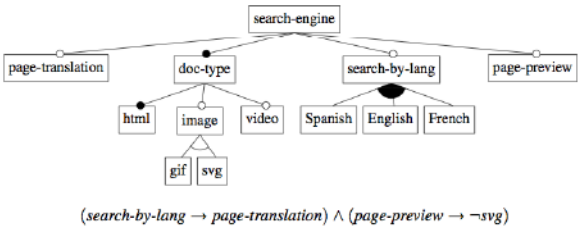
HYPOTHÈSE DE TRAVAIL 12 :
PROPRIÉTÉS DU PROCESSUS DE CONFIGURATION

Nous considérons dans nos travaux qu'il est nécessaire de s'assurer de la cohérence des configurations (voir Hypothèse 11) mais également de garantir des propriétés sur le processus de configuration. Il s'agit ainsi dans le cadre d'une LPL complexe, de supporter un processus de configuration qui *(i)* soit dynamique et interactif, *(ii)* permette à l'utilisateur de s'assurer de la cohérence de ses choix et *(iii)* guide l'utilisateur dans ses choix.

3.3.1.1 Logique booléenne

Ainsi, les FM – et plus particulièrement la sémantique qu'ils expriment sur les features – doivent être utilisés lors du processus de configuration. Schobbens *et al.* proposent ainsi une sémantique formelle applicable aux FM booléens (sans attributs) permettant de dériver à partir d'un FM une forme normale conjonctive (CNF) définissant l'ensemble des configurations autorisées du FM [Schobbens 07]. La Figure 3.1 montre ainsi un FM ainsi qu'une formule booléenne [1] qui lui correspond. Seules différentes combinaisons de features sont autorisées selon la sémantique du FM : il est alors possible de considérer les features comme des littéraux booléens considérés en conjonction.

Trouver des solutions au sein de la CNF représentant le FM revient alors à déterminer si en valuant les différents littéraux de la CNF, celle-ci peut être vraie. Cela revient à appliquer un algorithme de satisfiabilité (SAT).



(a) FM représentant un moteur de recherche

search-engine ∧ (page-translation → search-engine) ∧ (1)
(doc-type → search-engine) ∧ (2)
(search-by-lang → search-engine) ∧ (3)
(page-preview → search-engine) ∧ (html → doc-type) ∧ (4)
(image → doc-type) ∧ (video → doc-type) ∧ (5)
(Spanish → search-by-lang) ∧ (6)
(English → search-by-lang) ∧ (7)
(French → search-by-lang) ∧ (8)
(gif → image) ∧ (svg → image) ∧ (9)
(search-engine → doc-type) ∧ (doc-type → html) ∧ (10)
(search-by-lang → (Spanish ∨ English ∨ French)) ∧ (11)
(image) → (gif ∧ ¬svg ∨ ¬gif ∧ svg) ∧ (12)
(search-by-lang → page-translation) ∧ (13)
(page-preview → ¬svg) (14)

(b) Formule du FM

FIGURE 3.1 – Un FM et sa formule extraits de [Mendonca 09]

Mendonca *et al.* discutent dans leurs travaux de la complexité algorithmique des solveurs SAT [Mendonca 09]. En effet, les problèmes SAT sont des problèmes de classe NP-complet, cela signifie que rien ne permet de prouver théoriquement que quelques soient les entrées du problème, il sera possible de trouver un algorithme capable de résoudre ce problème en temps polynomial [Wilf 94]. Cependant, Mendonca *et al.* proposent des heuristiques pour utiliser les algorithmes SAT dans le cadre des FM et obtiennent des performances intéressantes même pour des FM de plus de 10 000 features. Ils montrent ainsi que ce type de solveur est effectivement utilisable dans le cadre des FM booléens.

HYPOTHÈSE DE TRAVAIL 13 :
COMPLEXITÉ THÉORIQUE DES ALGORITHMES ET MISE EN PRATIQUE

Nous considérons dans nos travaux qu'il est essentiel que les solutions proposées remplissent nos objectifs sur le plan théorique mais soient également utilisables en pratique. Les complexités des algorithmes sont donc un élément important de notre étude.

3.3.1.2 Logiques floues et modales

D'autres types de travaux proposent ne pas considérer les FM selon l'angle d'une pure logique booléenne.

1. Celle-ci n'est pas à la forme normale conjonctive dans notre illustration.

Bagheri *et al.* proposent ainsi de réaliser un processus de configuration des FM en considérant des exigences et des souhaits utilisateurs [Bagheri 10]. Ils proposent ainsi d'utiliser une logique floue afin de déterminer la configuration la plus adaptée intégrant les exigences et prenant en compte un maximum de souhaits utilisateurs, tout en acceptant les conflits avec certains souhaits. Ce type de logique a l'avantage de permettre une plus grande marge de manœuvre par les solveurs lors du raisonnement sur les FM. Il peut être cependant difficile pour l'utilisateur de la LPL de faire la distinction entre les exigences et les souhaits.

Dans une approche différente, Diskin *et al.* proposent d'encoder différemment les FM, sous la forme de structure de Kripke, afin de ne pas perdre la sémantique du FM au sein du produit [Diskin 13]. Ils postulent en effet que deux FM ayant des hiérarchies complètement différentes pourront avoir des configurations similaires. Une configuration dans une logique booléenne ne correspondant qu'à un ensemble de littéraux valués, sans aucune considération pour la hiérarchie d'origine du FM. Ils proposent ainsi une nouvelle sémantique formelle des FM, ainsi qu'un processus de configuration s'appuyant sur la logique modale.

HYPOTHÈSE DE TRAVAIL 14 :
FLEXIBILITÉ DU PROCESSUS DE CONFIGURATION

La logique du processus de configuration diffère de la structure du FM. Il nous semble important d'autoriser de la flexibilité entre le processus de configuration et la modélisation de la variabilité. Nous ne souhaitons pas que celle-ci limite le processus de configuration. En outre, nous ne prenons pas en compte dans nos travaux la notion de préférence de configuration : ce point reste une perspective de cette thèse.

3.3.2 Gestion des références et des cardinalités

Plusieurs travaux s'attachent à expliquer la configuration des FM proposant des mécanismes de références vers d'autres FM, ou permettant l'ajout d'attributs sur les features comme une cardinalité ou un type.

Czarnecki *et al.* discutent ainsi des mécanismes de configuration attachés au formalisme des FM qu'ils ont défini (voir sous-section 2.3.3) [Czarnecki 05b]. Les auteurs expliquent ainsi que des features et leur sous-arbre vont pouvoir être *clonés* lors de la configuration : cela signifie qu'une feature ayant une cardinalité supérieure à 1 pourra être sélectionnée plusieurs fois ; et si cette feature possède des enfants, ses enfants seront également disponibles plusieurs fois à la sélection.

Les références sont également gérées de manière à dupliquer le FM référencé en l'attachant à la feature qui le référence : un nouveau sous-arbre est donc créé qui est utilisé lors de la configuration. La Figure 3.2 illustre ce mécanisme.

Enfin les auteurs évoquent également la manière de donner une valeur à une feature lorsque celle-ci possède un type. Ils ne discutent cependant pas de la validation de la valeur spécifiée.

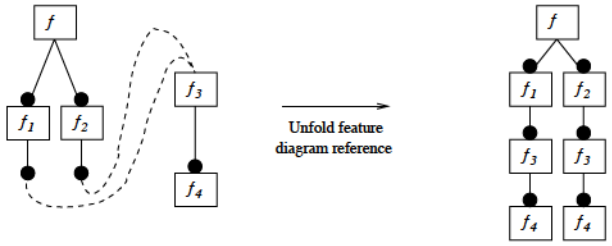


FIGURE 3.2 – Gestion des références lors de la configuration. Extrait de [Czarnecki 05b]

HYPOTHÈSE DE TRAVAIL 15 :
PROCESSUS DE CONFIGURATION BASÉ SUR LES FM ET LES MODÈLES

Si la gestion des cardinalités et des références nous semblent essentielles pour représenter la variabilité des LPL complexes, la sémantique et le support de ces types de variabilités tels que présentés dans la littérature ne sont pas en adéquation avec nos hypothèses de travail 8 et 12.
Il nous semble ainsi plus pertinent de conserver dans nos travaux un formalisme de FM plus traditionnel, en ne considérant que des FM booléens, et d'utiliser d'autres modèles pour le support des références et cardinalités, en les intégrant dans un processus de configuration qui les prend en compte.

3.4 Workflow de configuration

La définition de formalismes de raisonnement sur les FM, et d'outil en support à ces formalismes n'est pas suffisante pour permettre de réaliser un processus de configuration dans le cas de FM très complexes. Nous présentons dans cette section les travaux autour de la définition des *workflow de configurations* afin de préciser les différentes étapes et acteurs du processus de configuration.

3.4.1 Multi-Staged Configuration Process

Czarnecki *et al.* proposent une approche permettant de modulariser le processus de configuration en plusieurs étapes, le *Multi-Staged Configuration Process* (MSCP) [Czarnecki 05b]. L'idée est de définir un workflow de configuration dans lequel, à chaque étape, certaines options sont éliminées, réduisant le nombre de possibilités pour les étapes suivantes. Les différentes étapes peuvent être déterminées selon différents critères :

- en fonction du cycle de vie du produit (par ex. design, test, etc),
- en fonction des rôles des différents intervenants dans la configuration du produit (par ex. le responsable de la sécurité, le responsable marketing, etc)
- en fonction de la cible du produit (par ex. un composant configuré pour un système plus vaste ou pour être utilisé indépendamment).

Pour gérer le MSCP, les auteurs définissent deux opérations distinctes sur les FM : la *configuration* et la *spécialisation*. La **configuration** est définie comme l'ensemble des features sélectionnées en accord avec les contraintes définies sur le FM. Les auteurs précisent par ailleurs

que la relation entre une configuration et son FM est comparable à la relation entre une instance et sa classe en programmation orientée objet. La **spécialisation** est définie comme la transformation qui, à partir d'un FM, retourne un autre FM dans lequel l'ensemble des configurations représenté par le FM résultant est un sous-ensemble de l'ensemble des configurations du FM d'origine. Une spécialisation consiste donc à réduire les choix d'un FM.

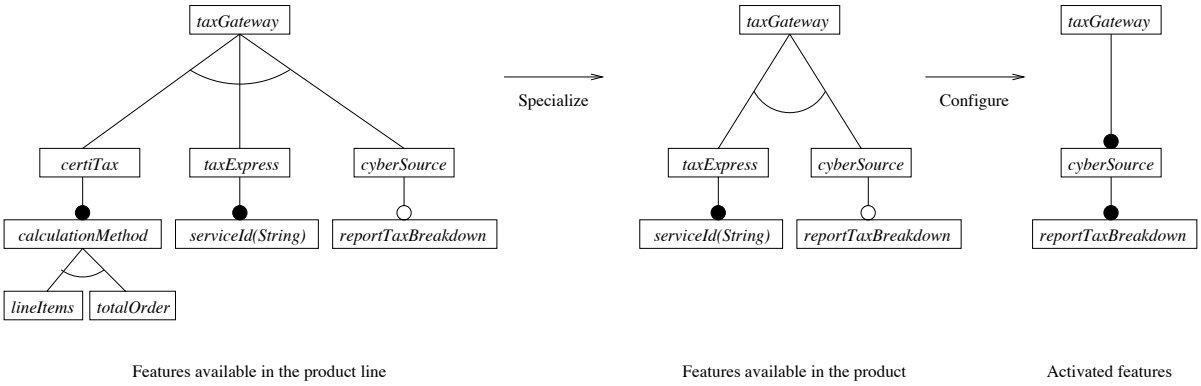


FIGURE 3.3 – Spécialisation et configuration d'un FM. Extrait de [Czarnecki 05b]

La Figure 3.3 montre un exemple de FM représentant un système de paiement de taxes d'abord spécialisé, puis configuré. On constate bien que le deuxième FM représenté, a été spécialisé : des choix sont encore possibles mais en moins grand nombre. Enfin, le troisième FM ne présente plus aucun choix, il s'agit d'une configuration finale. Il est important de noter par ailleurs que les FM spécialisés et configurés respectent parfaitement les contraintes du FM d'origine.

Dans les mêmes travaux, Czarnecki *et al.* définissent également le concept de configuration *multi-niveaux* [Czarnecki 05b]. Il s'agit là encore d'un processus de type MSCP, à la différence que chaque étape de configuration donnera lieu à une *configuration manuelle* qui sera répercutée comme une *spécialisation automatique* vers un FM présentant davantage de variabilité. Ainsi les acteurs réaliseront toujours une configuration complète, sans laisser de choix libres et ne verront qu'une partie du FM à chaque fois.

La Figure 3.4 illustre un exemple de configuration multi-niveaux en deux étapes sur un exemple de FM pour le paiement de taxe. On constate qu'au premier niveau (première colonne), un FM est configuré lors de la première étape (première ligne). La configuration obtenue est ensuite appliquée comme spécialisation au FM du niveau 1 (deuxième colonne) de manière automatique : on est alors toujours dans la première étape. La seconde étape (seconde ligne) consiste ainsi à configurer manuellement le FM spécialisé obtenu. On peut noter sur cet exemple que le premier FM au niveau 0 dispose d'un groupe OR alors que le premier FM au niveau 1 dispose d'un groupe XOR : le choix final du type de taxe est volontairement reporté au niveau 1.

Les travaux de Reiser *et al.* ont également porté sur la configuration de FM multi-niveaux [Reiser 06]. Dans leur cas cependant, la variabilité était directement définie par une hiérarchie de FM en lien les uns avec les autres, contrairement aux travaux de Czarnecki.

Par ailleurs, Classen *et al.* proposent dans leurs travaux de définir une sémantique formelle pour les processus de configuration multi-niveaux [Classen 09]. Ils définissent dans cette sémantique les concepts de niveaux et de chemin de configurations exploitant ces niveaux.

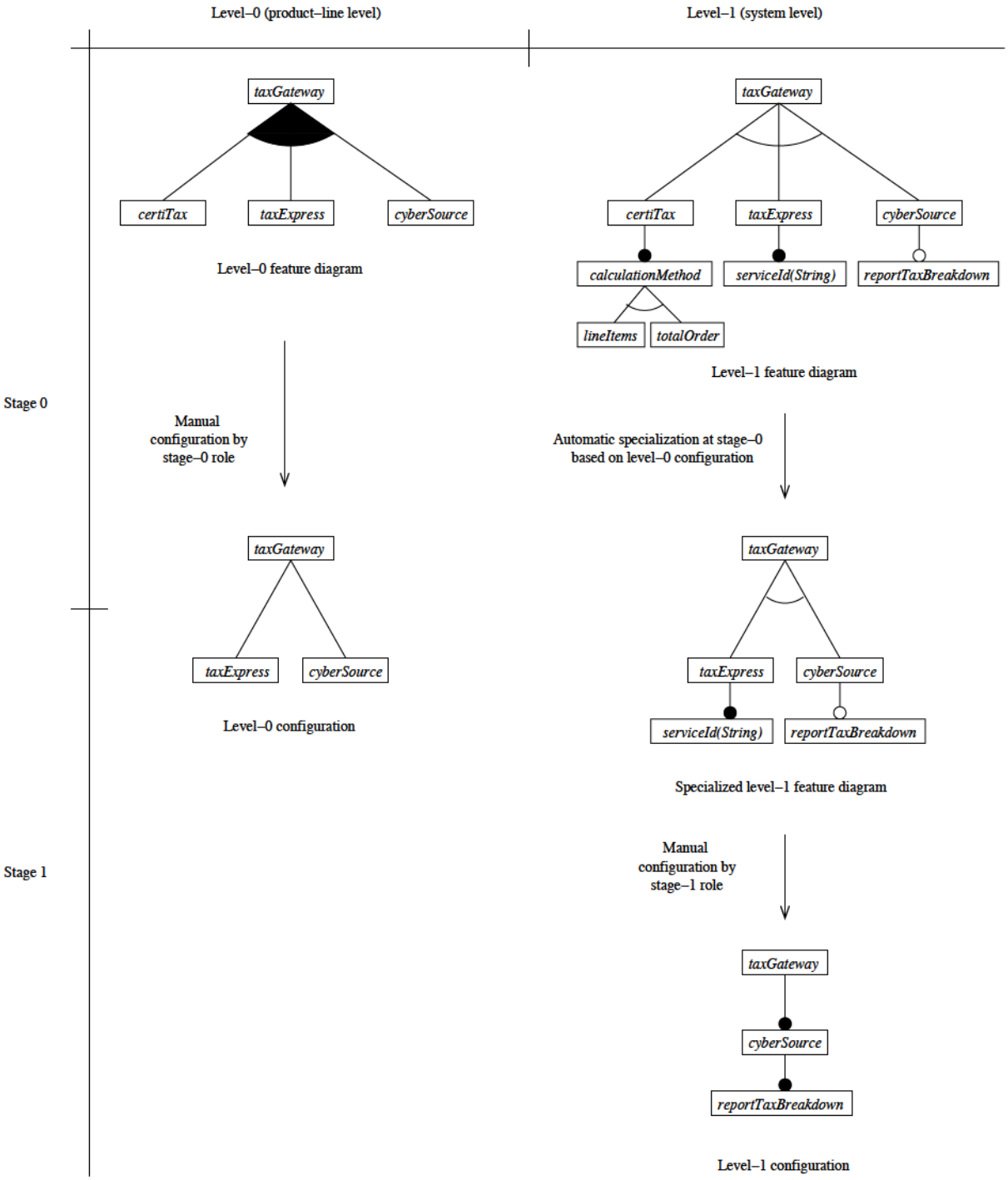


FIGURE 3.4 – Illustration d'une configuration multi-niveaux. Extrait de [Czarnecki 05b]

HYPOTHÈSE DE TRAVAIL 16 :
UN PROCESSUS DE CONFIGURATION PAR ÉTAPES

La définition d'un processus de configuration multi-niveaux fait écho à la séparation des préoccupations dans les modèles de variabilité (voir Hypothèse 6).
Il nous semble essentiel dans le cadre de nos travaux de conserver un processus de configuration décomposable en étapes de configuration.

3.4.2 Configuration Multi-Perspectives

Une des problématiques de la configuration des FM que l'on retrouve dans de nombreux travaux concerne la gestion de la complexité des FM : il est en effet difficile à un utilisateur d'effectuer des choix dans un FM de plusieurs milliers de features.

Les travaux de Schroeter *et al.* portent ainsi sur le concept de configuration multi-perspectives : l'idée est de déterminer des points de vue utilisateur sur le FM à configurer [Schroeter 12]. L'utilisateur ne verra ainsi qu'une partie du FM lors de la configuration, ce qui lui évite de se perdre dans la complexité du FM complet. Des features pouvant être partagées par plusieurs points de vue, les auteurs proposent un mécanisme permettant la composition des vues tout en garantissant la conformité avec le FM d'origine.

Hubaux *et al.* définissent quant à eux une sémantique formelle permettant de réaliser automatiquement des vues utilisateurs de configuration sur un FM en fonction des préoccupations exprimées [Hubaux 13].

HYPOTHÈSE DE TRAVAIL 17 :
PRISE EN COMPTE DES PERSPECTIVES LORS DU PROCESSUS

La notion de perspectives lors du processus de configuration est une caractéristique qu'il nous semble important de conserver dans le cadre de la configuration des systèmes-de-systèmes. Ainsi un sous-système pourra être configuré selon sa perspective locale ou dans le contexte global d'un système-de-systèmes.

3.4.3 Feature Configuration Workflows

Hubaux, impliqué dans les travaux sur les processus de configuration multi-niveaux et multi-perspectives, va plus loin en définissant de manière formelle une modélisation de workflows de configuration complexes [Hubaux 12b].

Le processus de configuration peut être une tâche complexe, en particulier si plusieurs utilisateurs sont impliqués. L'utilisation de techniques comme les configurations multi-niveaux ou multi-perspectives offrent des résultats intéressants mais méritent selon Hubaux d'être combinés.

Il propose ainsi de réutiliser le langage de workflow YAWL afin de définir les différentes tâches de configuration pour chaque acteur impliqué, en spécifiant une perspective sur le FM pour chaque tâche. Différentes tâches pouvant s'appliquer en parallèle, il définit également un mécanisme de réconciliation permettant de gérer les conflits au sein de son workflow.

La Figure 3.5 montre ainsi un exemple de workflow de configuration pour une LPL complexe. On constate dans cet exemple la présence de nombreuses tâches de configuration distinctes s'appliquant sur des perspectives différentes d'un FM. Par ailleurs certaines tâches sont à réaliser en parallèle.

La définition d'un workflow de configuration permet de prévoir et d'organiser de manière cohérente la configuration de FM complexes. Cependant, la notion de workflow implique d'être capable de planifier le processus de configuration. Or Botterweck décrit le processus de configuration dans le cadre d'une LPL pour des systèmes-de-systèmes de la façon suivante :

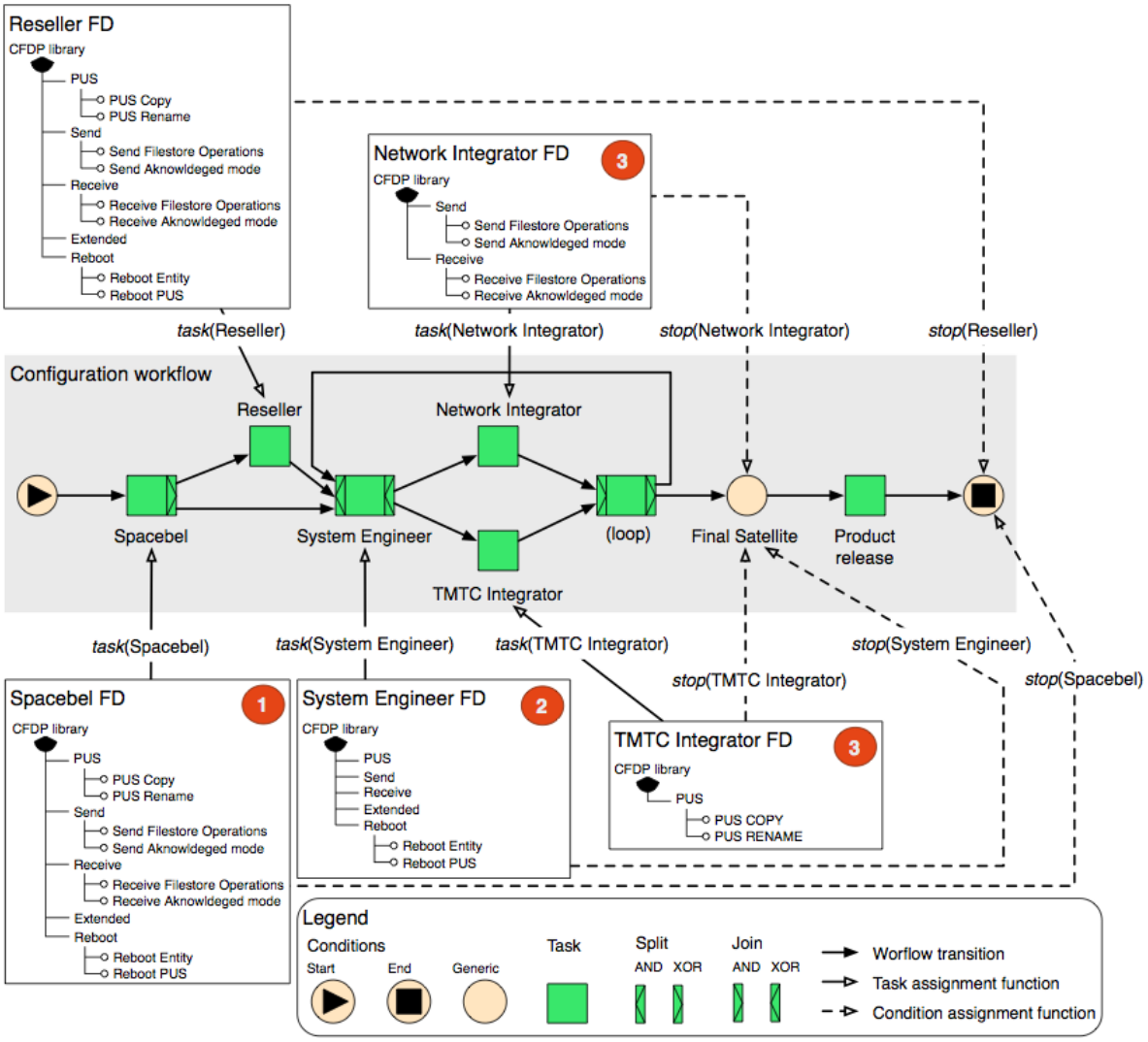


FIGURE 3.5 – Illustration d'un workflow de configuration. Extrait de [Hubaux 09]

> *Depending on the scenario and the power to "dictate" conditions, this occurs in various directions :*
>
> — *System to subsystem [...]*
> — *System to supersystem [...]*
> — *System to a neighboring system on a similar level [...]*
>
> [BOTTERWECK 13]

Par ailleurs des travaux réalisés dans le domaine des workflows pour le travail coopératif discutent également des bénéfices apportés par des workflows offrant de la flexibilité dans l'agencement des tâches [Abbott 94].

HYPOTHÈSE DE TRAVAIL 18 :
ABSENCE D'ORDRE PRÉDÉFINI DANS LE PROCESSUS

Nous postulons que dans le cadre de la réalisation d'une configuration pour un système-de-systèmes l'ordre du processus de configuration ne doit pas être imposé par les outils de raisonnement ni par la modélisation de la variabilité (voir Hypothèse 14).

3.5 Configuration dans les lignes de produits multiples

Comme le relatent Höll *et al.* dans leur état de l'art sur les lignes de produits multiples, les différentes approches exposées ci-dessus peuvent être réutilisées dans le cadre des LPLM [Holl 12]. En particulier l'approche de Hubaux peut être utilisée sur une LPLM : au lieu d'exploiter différentes perspectives sur un même FM, ce sont les différents FM issus de la LPLM qui seront utilisés.

Par ailleurs, Höll *et al.* insistent dans leur étude sur l'importance des approches de configuration collaboratives dans le cadre des lignes de produits multiples : plusieurs acteurs peuvent en effet contribuer en même temps à une configuration très complexe. On peut citer par exemple l'approche de Mendonça *et al.* qui propose la définition d'un modèle de configuration collaboratif [Mendonça 08]. Leurs travaux sont proches de ceux sur les configurations multi-perspectives : il s'agit de définir différentes vues sur un FM à partir desquelles des graphes de dépendances sont calculés pour déterminer la manière dont les vues sont inter-reliées selon la sémantique du FM. La configuration des vues peut alors se dérouler en parallèle, des mécanismes permettant de fusionner les résultats étant apportés par la solution.

Rabiser *et al.* proposent une approche différente de la configuration collaborative dans le cadre d'une ligne de produits multiple en reprenant le principe du patron publish/subscribe [Rabiser 10b]. L'idée de leur approche est de réaliser localement des actions de configuration et de publier ces actions à travers un "board" afin de les faire valider. Lorsqu'elles sont acceptées, les actions de configuration sont considérées comme acquises pour les autres collaborateurs. Par ailleurs les différents collaborateurs ont la possibilité de changer certaines décisions. La Figure 3.6 illustre le principe de l'approche proposée.

Enfin Dhungana *et al.* proposent également une approche originale de la configuration des lignes de produits multiples à travers l'approche *Invar*, en considérant que chaque LPL est définie avec son propre mécanisme de configuration et ses propres informations de dépendance avec les autres LPL [Dhungana 11]. Ainsi la configuration se fait à travers une interface capable de communiquer avec plusieurs services de configurations pour les différentes LPL impliquées dans la LPLM. Des informations d'inter-dépendances sont également exploitées durant ce processus de configuration. L'architecture de l'approche est illustrée dans la Figure 3.7.

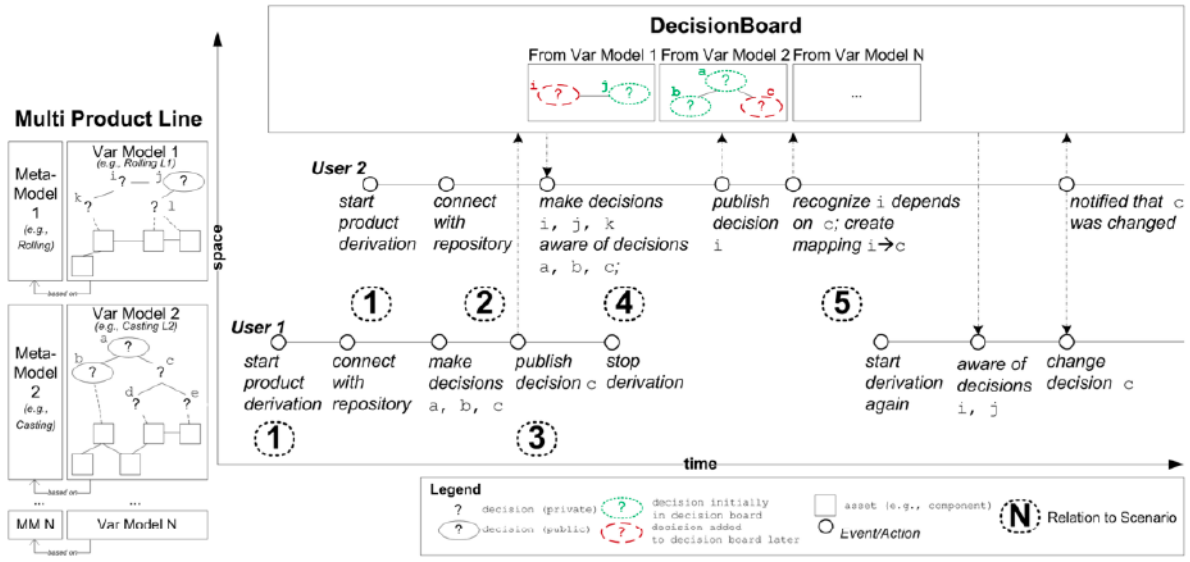


Figure 1. The DecisionBoard enables awareness in product derivation in multi product lines. In the example two users perform product derivation for two (sub-)systems defined by two variability models that are based on different meta-models. They make decisions locally and publish selected decisions to the decision board. Users can consult the decision board about decisions shared by others. For instance, user 2 creates a mapping from decision i (e.g., rolling mode in the mini-mill) to c (e.g., number of strands in the caster) to be notified after changes to the number of strands.

FIGURE 3.6 – Illustration d'un processus de configuration collaboratif. Extrait de [Rabiser 10b]

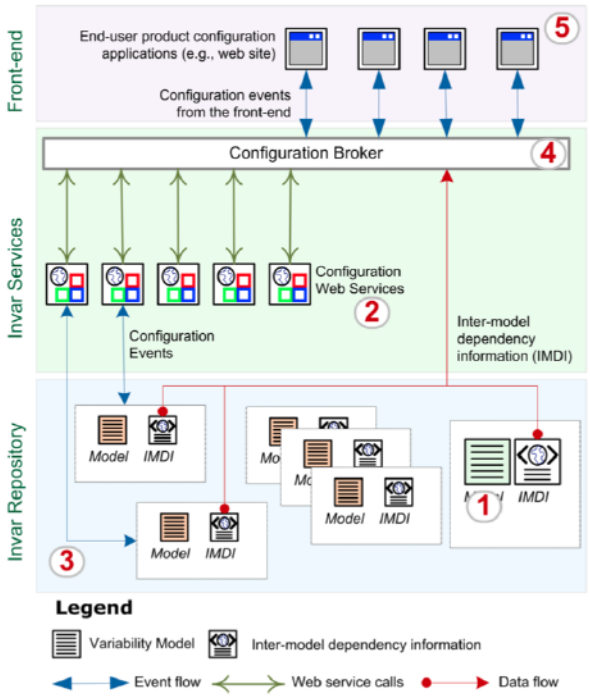


FIGURE 3.7 – Illustration de l'architecture de configuration dans la solution *Invar*. Extrait de [Dhungana 11]

HYPOTHÈSE DE TRAVAIL 19 :
PRISE EN COMPTE DES DÉPENDANCES ENTRE LES FM

L'utilisation des dépendances *entre* les modèles de variabilité est essentielle au processus de configuration, dans le cadre d'une LPL complexe dont la variabilité est modélisée en respectant la séparation des préoccupations. Nous considérons, par ailleurs, l'approche collaborative du processus de configuration comme une perspective de cette thèse.

3.6 Conclusion

Nous avons réalisé dans ce chapitre un état de l'art sur les processus de configuration des LPL en mettant en avant les différentes hypothèses de travail que nous souhaitons utiliser dans nos travaux.

Nous nous intéressons en particulier aux travaux les plus avancés de définition des processus de configuration, tels que les travaux sur les workflows de configuration ou sur les processus collaboratifs.

CHAPITRE 4

POSITIONNEMENT ET OBJECTIFS

Sommaire

Résumé Sur la base des défis et des hypothèses de travail présentées dans les chapitres précédents, nous établissons dans ce chapitre nos objectifs détaillés et les positionnons par rapport à l'existant.

4.1 Introduction

Nous revenons dans ce chapitre sur les différentes limitations observées dans les travaux sur l'état de l'art relatif aussi bien à la modélisation des lignes de produits logiciels complexes que des processus de configurations flexibles dans ces mêmes LPL complexes. Nous définissons dans le même temps de manière explicite les objectifs que nous souhaitons atteindre dans le cadre de nos travaux. Puis nous synthétisons quelques travaux représentatifs de l'état de l'art relativement aux objectifs que nous avons définis précédemment.

4.2 Conserver la séparation des préoccupations et des formalismes dans la modélisation des LPL complexes

Nous avons présenté dans le chapitre 2 les différents travaux relatifs à la modélisation de la variabilité dans les LPL en nous intéressant notamment à la modélisation des LPL complexes.

Nous avons ainsi précisé que nous nous intéressons à l'ingénierie du domaine d'une LPL dont les produits sont des systèmes-de-systèmes (Hypothèse 1) et que nous nous concentrons uniquement dans cette thèse sur l'étape de modélisation de la variabilité au sein de l'ingénierie du domaine (Hypothèse 2).

Notre premier objectif est donc de permettre la modélisation de la variabilité des LPL complexes, tout en conservant une bonne séparation des préoccupations et des formalismes. Cet objectif vise à répondre à notre premier défi : "*Comment modéliser de manière cohérente la variabilité d'un système-de-systèmes au sein d'une LPL ?*".

Nous définissons dans la suite les différents sous-objectifs relatifs à ce défi.

4.2.1 Représentation d'une LPL par l'utilisation d'un modèle du domaine et de modèles de variabilité

Il est nécessaire de considérer une variabilité intégrant plusieurs dimensions pour représenter la variabilité d'une LPL dédiée à des systèmes-de-systèmes (Hypothèses 4 et 9). Ainsi la séparation des préoccupations et l'exploitation de plusieurs modèles de variabilité nous semblent être des approches intéressantes dans notre contexte (Hypothèses 6 et 8).

Par ailleurs des recherches ont déjà été menées sur ces thématiques (Hypothèse 10) que nous souhaitons poursuivre dans nos travaux en intégrant notamment des multiplicités plus libres (Hypothèse 6).

4.2.2 Garantir la cohérence de la LPL

Nous avons vu dans notre état de l'art que les modèles de variabilité pouvaient rapidement devenir complexes, ce qui rend difficile leur analyse. Ainsi nous souhaitons dans nos travaux pouvoir assurer la cohérence de la variabilité d'une LPL complexe (Hypothèse 5) aussi bien de manière théorique, que de manière pratique (Hypothèse 2).

4.2.3 Exploiter des standards et le point de vue de l'utilisateur final pour la définition de la LPL

L'état de l'art nous a montré qu'il existe de nombreuses solutions pour représenter la variabilité d'une LPL. Il nous semble ainsi important de proposer dans nos travaux une solution qui

réutilise les standards de la communauté des lignes de produits logiciels et de l'industrie logicielle (Hypothèse 8). En outre, nos choix de modélisation de la variabilité se veulent également pragmatiques, et fonction des outils disponibles permettant leur utilisation (Hypothèse 15).

Par ailleurs, même s'il nous semble important dans nos travaux de réutiliser les standards, la conservation du point de vue de l'utilisateur final lors de la définition des modèles de variabilité est également l'un de nos objectifs (Hypothèse 7).

4.3 Garantir la flexibilité et la cohérence du processus de configuration

Notre second défi concerne l'ingénierie de l'application des LPL complexes : "*Comment permettre un processus de configuration flexible et cohérent dans le cadre d'une LPL complexe ?*". Nous n'abordons en effet dans nos travaux sur l'ingénierie de l'application que la problématique du processus de configuration (Hypothèse 3). Nous avons présenté dans le chapitre 3 un état de l'art concernant les processus de configuration des LPL. Nous nous appuyons sur celui-ci et les hypothèses de travail que nous avons définies pour résoudre ce défi dans notre deuxième objectif.

Nous proposons ainsi les sous-objectifs suivants.

4.3.1 Permettre la réalisation d'un processus de configuration dynamique et cohérent dans la LPL

Comme nous l'avons présenté dans notre état de l'art, la cohérence de la configuration et du processus de configuration sont essentiels, en particulier dans le cas d'une LPL complexe (Hypothèses 11 et 12). Il nous semble également important d'exploiter des dépendances exprimées *entre* les modèles de variabilité afin de s'assurer de la cohérence du processus (Hypothèse 19).

Par ailleurs, les travaux de l'état de l'art montrent la nécessité d'avoir un processus qui soit également dynamique afin d'offrir à l'utilisateur un feedback immédiat sur ses choix durant le processus de configuration (Hypothèse 12). Permettre un processus dynamique est donc aussi un objectif de nos travaux.

4.3.2 Permettre la gestion d'un processus de configuration par étapes : annulation et reprise de configuration

Comme nous le montrent les travaux de l'état de l'art, gérer un processus de configuration complexe nécessite de pouvoir décomposer ce processus en étapes selon le fameux principe du "*divide and conquer*" (Hypothèse 16). Ainsi nos propres travaux doivent proposer un processus de configuration par étapes.

En outre, nous ne pouvons pas concevoir un processus de configuration dans lequel l'utilisateur ne voudrait jamais revenir sur ses choix : il nous semble essentiel de proposer dans nos travaux des solutions pour permettre d'annuler certaines décisions. De même, si nous ne nous concentrons pas dans nos travaux sur une approche collaborative du processus de configuration (Hypothèse 19), il nous paraît important de proposer une solution permettant de considérer une configuration terminée ou partielle afin de réaliser une nouvelle configuration.

4.3.3 Permettre la réalisation d'un processus de configuration dans une LPL sans ordre prédéfini

Nous avons montré dans notre état de l'art que le processus de configuration ne devait pas nécessairement être dépendant de la modélisation de la variabilité (Hypothèse 14). Par ailleurs les travaux les plus avancés de l'état de l'art des processus de configuration concernent les workflows de configuration, dont le but est de planifier les différentes étapes du processus. Nous avons postulé dans notre état de l'art, que cette planification n'était pas nécessairement idéale dans le cadre des LPL complexes destinées à la réalisation de systèmes-de-systèmes (Hypothèse 18). Un des objectifs de notre thèse est donc de permettre un processus de configuration cohérent au sein d'une LPL complexe sans ordre prédéfini.

4.4 Garantir l'utilisabilité du processus de configuration

Dans la lignée du point de vue d'Arrango sur l'ingénierie du domaine [Arango 89], nous souhaitons dans cette thèse proposer une approche pragmatique, réalisable et utilisable du processus de configuration d'une LPL complexe afin de répondre complètement à notre second défi (Hypothèse 2). Nous proposons ainsi les sous-objectifs suivants.

4.4.1 Permettre le guidage des utilisateurs durant le processus de configuration

Comme nous l'avons montré dans notre état de l'art, les différentes contributions sur les processus de configuration visent à réduire la complexité de cette activité pour l'utilisateur final. Le guidage de l'utilisateur durant le processus de configuration est donc primordial afin de l'aider dans sa tâche (Hypothèse 12). Par ailleurs, afin de réaliser ce guidage, des travaux ont été menés sur la définition de perspectives de configuration : nous souhaitons exploiter dans nos propres travaux ces notions afin de contextualiser les choix utilisateurs (Hypothèse 17).

4.4.2 Garantir des temps de réponse et une empreinte mémoire raisonnable pour l'implémentation d'une solution

Nous avons discuté dans notre état de l'art le fait que la configuration des FM était encore un problème de décision difficile à l'heure actuelle, comparable à un problème de satisfiabilité. Il nous apparaît ainsi essentiel, dans le cadre de nos travaux exploitant ce type de mécanisme, de pouvoir offrir des garanties quant à l'utilisabilité concrète des différents algorithmes et outils que nous définissons, aussi bien sur le plan des temps de réponses, qu'au niveau de l'empreinte mémoire des implémentations proposées (Hypothèse 13).

L'un de nos objectifs sera donc de garantir ces propriétés dans nos contributions.

4.5 Synthèse

Nous proposons dans le Tableau 4.1 une synthèse des travaux de l'état de l'art qui nous semblent les plus proches des objectifs que nous avons précédemment définis.

Nous avons représenté dans ce tableau les sous-objectifs par rapport à leur numéro de sous-section. Nous utilisons dans le tableau la légende suivante :

— ✓ représente un sous-objectif complètement atteint,
— ✗ représente un sous-objectif pas du tout atteint,
— - représente un sous-objectif partiellement atteint,
— et N.P. représente un sous-objectif qui n'est pas pertinent pour les travaux considérés.

	Objectif 1			Objectif 2			Objectif 3	
Réference	4.2.1	4.2.2	4.2.3	4.3.1	4.3.2	4.3.3	4.4.1	4.4.2
[Pohl 05]	✗	✗	-	-	✗	-	N.P	N.P.
[Czarnecki 02]	✗	✗	-	-	-	✗	✓	N.P.
[Bąk 11]	✓	✓	✗	N.P.	N.P.	N.P.	N.P	N.P.
[Rosenmüller 10]	-	✗	✓	✓	✗	✗	✓	N.P.
[Dhungana 11]	-	N.P.	✓	✓	-	✗	✓	✓
[Rabiser 10b]	-	N.P.	✓	✓	-	✓	✗	N.P.
[Hubaux 09]	-	✓	✓	✓	-	✗	✓	-

TABLE 4.1 – Synthèse des travaux de l'état de l'art relativement aux objectifs

Peu de travaux de l'état de l'art traitent à la fois de la modélisation de la variabilité et du processus de configuration, en particulier dans le cadre de systèmes complexes. À notre connaissance, aucun travail ne répond à l'ensemble des objectifs que nous avons fixés pour la modélisation et l'utilisation d'une LPL destinée à la réalisation de systèmes-de-systèmes.

Nous présentons dans la suite de ce document nos contributions (Partie II) puis nous montrons comment elles valident l'ensemble des objectifs présentés ici (Partie III).

Deuxième partie

Contribution

NOTATIONS

Le formalisme que nous définissons utilise de nombreux concepts et notations. La plupart sont des notations usuelles et le lecteur les reconnaîtra immédiatement. Cependant certaines d'entre elles sont définies tout au long du formalisme et sont réutilisées plus loin.

Nous avons regroupé ici les différents éléments récurrents de notations. Le lecteur pourra ainsi revenir à cette page pour retrouver certains éléments.

Lettres employées dans le formalisme

— $\mathcal{A}$: une association (Définition 5.2.4)
— $\mathcal{A}[fm]$: une action système sur un FM (Définition 5.4.1)
— $\mathcal{AS}$: une action système de manière générale (Définition 6.5.1)
— $\mathcal{C}$: un contexte (Définition 6.3.1)
— $\mathcal{CC}$: une configuration composite (Définition 5.5.3)
— $\mathcal{CPS}$: une étape du processus de configuration (Définition 6.2.1)
— $\mathcal{CST}$: une contrainte d'un FM (Définition 5.3.2)
— $\mathcal{D}$: un concept du domaine (Définition 5.2.3)
— $\mathcal{E}$: une extrémité d'association (Définition 5.2.4)
— $\mathcal{F}$: une feature (Définition 5.3.1)
— $\mathcal{G}$: un groupe de features au sein d'un FM (Définition 5.3.1)
— $\mathcal{GCC}$: un gestionnaire de configuration composite (Définition 6.3.2)
— $\mathcal{H}$: un historique des actions utilisateur (Définition 6.5.4)
— $\mathcal{L}$: un lien (Définition 5.5.2)
— $\mathcal{M}$: un modèle du domaine (Définition 5.2.1)
— $\mathcal{Q}$: une action utilisateur (Définition 6.5.2)
— $\mathcal{S}$: un état de configuration (Définition 5.3.4)
— $\mathcal{SC}$: une sous-configuration (Définition 5.5.1)
— $\mathcal{T}$: une étape du processus de dérivation (Définition 6.5.3)
— $\mathcal{U}$: une multiplicité (Définition 5.2.2)
— $\mathcal{V}$: un feature model (Définition 5.3.3)

Symboles employés dans le formalisme

Utilisation du symbole $\varnothing$

Nous utilisons le symbole $\varnothing$ dans notre formalisme toutes les fois où nous avons besoin de représenter le fait qu'un élément peut avoir une valeur nulle ou non définie.

Utilisation du symbole $\rightsquigarrow$

Le symbole $\rightsquigarrow$ est employé de nombreuses fois dans notre formalisme entre différents types d'éléments. Il signifie une relation entre des éléments : il peut s'agir d'une relation d'instanciation, de référence ou encore de compatibilité.

- Entre une association $\mathcal{A}$ et les concepts $\mathcal{D}_x$ et $\mathcal{D}_y$ qu'elle relie : $\mathcal{A} \rightsquigarrow \{\mathcal{D}_x, \mathcal{D}_y\}$ (Définition 5.2.4)
- Entre un FM $\mathcal{V}$ et le concept $\mathcal{D}$ dont il représente la variabilité : $\mathcal{V} \rightsquigarrow \mathcal{D}$ (Définition 5.3.3).
- Entre un couple d'états de configuration $\mathcal{S}_x$ et $\mathcal{S}_y$ et la fonction de restriction $\mathcal{RF}$ par rapport à laquelle ils sont compatibles : $(\mathcal{S}_x, \mathcal{S}_y) \rightsquigarrow \mathcal{RF}$ (Propriété 5.4.11).
- Entre une paire d'états de configuration $\mathcal{S}_x$ et $\mathcal{S}_y$ et l'association $\mathcal{A}$ par rapport à laquelle ils sont compatibles : $\{\mathcal{S}_x, \mathcal{S}_y\} \rightsquigarrow \mathcal{A}$ (Propriété 5.4.12).
- Entre une sous-configuration $\mathcal{SC}$ et le concept qu'elle instancie $\mathcal{D}$: $\mathcal{SC} \rightsquigarrow \mathcal{D}$ (Définition 5.5.1).

Utilisation du symbole $\rightleftharpoons$

On utilise le symbole $\rightleftharpoons$ pour exprimer la compatibilité entre deux CPS (cette relation est bijective) : $\mathcal{CPS}_i \rightleftharpoons \mathcal{CPS}_j$ (Propriété 6.2.1).

Utilisation du symbole $[\![]\!]$

On utilise le symbole $[\![\ ]\!]$ exclusivement autour d'un FM $\mathcal{V}$ ou d'un état de configuration afin d'exprimer l'ensemble des configurations qu'il représente : $[\![\mathcal{V}]\!]$ (Définition 5.3.5).

Utilisation du symbole $\rightarrow$

On utilise le symbole $\rightarrow$ dans notre formalisme afin de symboliser une transformation à la suite de l'application d'une ou plusieurs actions.

- Il peut s'agir de la transformation d'un état de configuration suite à l'application d'une unique action (Définition 5.4.1), d'une règle de restriction (Propriété 5.4.7) ou encore d'une fonction de restriction (Propriété 5.4.8).
- Il peut s'agir de la transformation d'un GCC suite à l'application de l'algorithme de propagation (sous-sous-section 6.4.2.3), de l'application de l'algorithme d'annulation (Propriété 6.5.1), de l'application d'une action système (Définition 6.5.1), ou encore de l'application d'une action utilisateur (Définition 6.5.2).

La notation est également utilisée afin de symboliser l'orientation des règles et fonctions de restrictions.

Choix liés au formalisme

Nous distinguons dans notre formalisme les notions de *définition* et *propriété* de la manière suivante :

- une *définition* permet de formaliser un concept, lui même éventuellement composé d'autres concepts ;
- une *propriété* est utilisée aussi bien pour formaliser les propriétés existantes sur et entre les concepts définis, que pour définir les opérations applicables sur ces mêmes concepts.

Cette distinction est issue de la vision métamodèle du formalisme dans laquelle les classes identifiées au niveau du métamodèle sont introduites par une définition tandis que les autres éléments sont définis par une propriété.

Par ailleurs, afin de simplifier la lecture du formalisme nous avons fait le choix de ne pas préciser systématiquement les domaines d'application : ceux-ci sont implicitement déduits des concepts utilisés.

Ainsi la Définition 5.2.3 pourrait être écrite comme suit :

Soit Id un ensemble d'identifiants représentés par une chaîne de caractères, $\mathbb{V}$ l'ensemble des feature models satisfaisant à la Définition 5.3.3 et $\mathbb{U}$ l'ensemble des multiplicités satisfaisant à la Définition 5.2.2. On définit $\mathcal{D}$ un concept du domaine par le triplet : $\langle id, \mathcal{V}, \mathcal{U} \rangle$ avec

— $id \in Id$
— $\mathcal{V} \in \mathbb{V}$
— $\mathcal{U} \in \mathbb{U}$

CHAPITRE 5

MODÉLISER DES LIGNES DE PRODUITS COMPLEXES

Sommaire

Résumé Dans le cadre d'une LPL complexe, la variabilité est présente à la fois en terme de fonctionnalités mais également en terme de combinaisons de ces fonctionnalités.

Nous formalisons dans ce chapitre la variabilité d'une LPL complexe sous la forme d'un modèle du domaine exposant des concepts, associations et multiplicités. La variabilité en terme de fonctionnalités est représentée dans notre approche sous forme d'un unique FM par concept de la LPL. Nous définissons en outre l'expression des contraintes entre les différents FM reposant sur des règles et des actions de configuration.

Nous formalisons également la représentation d'une configuration dite "composite" qui constitue la configuration d'un produit pour une LPL complexe. Enfin nous terminons en définissant les propriétés de cohérence qui sont requises par notre modèle du domaine afin que celui-ci puisse être exploité au sein d'un processus de configuration.

Nous tentons ainsi de répondre dans ce chapitre au premier objectif de cette thèse, à savoir conserver la séparation des préoccupations et des formalismes dans la modélisation d'une LPL complexe (voir section 4.2).

5.1 Introduction

Une des activités de l'ingénierie du domaine identifiées par Arango s'attache à réaliser un modèle du domaine [Arango 89]. Dans le cadre des LPL, ce modèle doit représenter la variabilité de la famille de produits que l'on souhaite pouvoir concevoir. Cependant, la variabilité est aussi bien présente du point de vue des fonctionnalités, qu'au niveau de la manière dont ces fonctionnalités se combinent [van Ommering 02].

En conséquence, nous proposons un modèle du domaine capable de représenter la variabilité sous ces deux points de vue : un point de vue présentant les différents concepts de la LPL et la manière dont ceux-ci peuvent se combiner (section 5.2) ; et un point de vue axé sur les fonctionnalités, dans lequel la variabilité des différents concepts est représentée à un grain beaucoup plus fin, celui de la feature (section 5.3).

L'utilisation de plusieurs modèles de variabilité au sein d'un modèle du domaine permet davantage d'expressivité mais nécessite d'exprimer les relations entre ces différents modèles de variabilité. Nous définissons donc un troisième modèle qui a en charge de supporter cet aspect (section 5.4).

Sur la base de cette modélisation de la LPL, nous définissons le concept de "configuration composite" (section 5.5) avec pour objet de supporter toute la variabilité proposée par la ligne.

Finalement, nous montrons comment valider la cohérence de l'ensemble des informations de notre modèle du domaine, cohérence indispensable à la réalisation d'une configuration valide au sein de la LPL ainsi définie (section 5.6).

5.2 Représenter le domaine métier de la LPL

Dans cette section nous définissons formellement la notion de modèle du domaine. Nous nous appuyons sur ce formalisme dans l'ensemble de notre contribution.

5.2.1 Modèle du domaine

Nous définissons le **modèle du domaine** comme un ensemble de **concepts** et un ensemble d'**associations** entre ces concepts.

Le terme de concept est volontairement vague en ce sens qu'un concept peut représenter des éléments de différentes natures en fonction de la LPL que l'on souhaite modéliser. Il peut ainsi s'agir d'un sous-domaine d'un domaine complexe, mais cela peut être également une représentation d'un ensemble de composants techniques réunis sous la même étiquette. De manière générale, nous représentons comme un concept un ensemble d'éléments qui peuvent être définis au sein d'un même modèle de variabilité.

Les associations définissent la manière dont ces différents concepts interagissent : elles représentent donc la façon dont les différentes instances des concepts peuvent se combiner.

Définition 5.2.1 (Modèle du domaine) :
Nous définissons un modèle du domaine $\mathcal{M}$ comme un couple $\langle SC, SA \rangle$ où SC est un ensemble $\{\mathcal{D}_0, ..., \mathcal{D}_n\}$ de concepts et SA un ensemble $\{\mathcal{A}_0, ..., \mathcal{A}_m\}$ d'associations.

Exemple 1 : Esquisse du modèle du domaine

Nous définissons un immeuble comme une entité composée d'appartements. Chaque appartement va lui même être constitué de pièces et d'ouvertures. Les ouvertures peuvent être internes aux pièces de l'appartement (par exemple, une porte entre deux pièces) ou être partagées avec l'extérieur de l'appartement (une porte ou une fenêtre donnant sur l'extérieur).

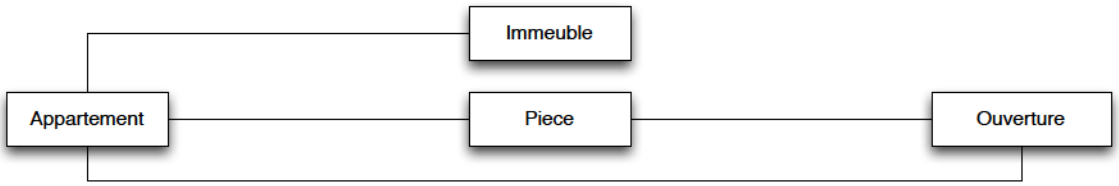


FIGURE 5.1 – Exemple simplifié de modèle du domaine pour l'exemple fil rouge

Nous montrons dans la Figure 5.1 un modèle du domaine simplifié pour notre exemple fil rouge. Elle représente les concepts : *Immeuble*, *Appartement*, *Pièce* et *Ouverture*, ainsi que les 4 associations qui les relient :

1. entre *Immeuble* et *Appartement* : l'immeuble est composé d'appartements,
2. entre *Appartement* et *Pièce* : un appartement est constitué de pièces,
3. entre *Pièce* et *Ouverture* : chaque pièce possède des ouvertures,
4. et entre *Appartement* et *Ouverture* : un appartement est accessible par des ouvertures extérieures.

5.2.2 Multiplicité

Nous avons besoin d'exprimer au sein du modèle du domaine des multiplicités sur les concepts et sur les extrémités des associations afin de décrire le nombre de fois qu'un concept peut être construit, et le nombre de fois qu'un concept peut être lié à un autre dans le produit final.

Exemple 2 : Illustration des multiplicités

Un produit de notre LPL fil rouge ne constitue qu'un seul immeuble, mais plusieurs appartements.

Une **multiplicité** est définie dans notre formalisme comme un intervalle entier. Elle possède une limite inférieure entière, qui peut être 0 ou 1 et une limite supérieure qui peut soit valoir exactement 1 (on dit alors qu'elle est bornée), soit être non bornée : elle est alors représentée par le symbole $*$.

Définition 5.2.2 (Multiplicité) :
Nous définissons une multiplicité $\mathcal{U}$ comme un couple $\langle L^-, L^+ \rangle$ avec $L^- \in \{0, 1\}$ et $L^+ \in \{1, *\}$.

Nous limitons volontairement la définition des multiplicités dans notre formalisme afin de garantir des propriétés de terminaison et de cohérence sur les algorithmes exploitant ce formalisme. L'extension de l'expressivité du formalisme fait partie de nos perspectives. Elle suppose, en effet, d'introduire les cardinalités fixées (par exemple, `1..2`) dans la définition des algorithmes ce qui complexifie nettement les preuves, en particulier dans le cas de la vérification de la réalisabilité.

On dit que la multiplicité est respectée si la cardinalité de l'ensemble [1] des éléments sur lequel porte cette multiplicité est contenue dans l'intervalle qu'elle définit.

Propriété 5.2.1 (Respect d'une multiplicité) :
Soit $\mathcal{U} = \langle L^-, L^+ \rangle$ une multiplicité et $S = \{e_0, ..., e_n\}$ un ensemble d'éléments sur lequel porte cette multiplicité. On note la cardinalité de l'ensemble S avec la notation : $|S|$ et on considère $|\emptyset| = 0$.

On dit que $\mathcal{U}$ est respectée pour S si :

1. $|S| \geqslant L^-$ et
2. $|S| \leqslant L^+$ en considérant $* = +\infty$.

On note alors $|S| \in \mathcal{U}$.

Les multiplicités sur les concepts vont permettre de déterminer le nombre de fois que le concept peut être instancié, par l'utilisateur. Celles sur les associations vont permettre de déterminer le nombre de fois qu'une instance d'un concept pourra être liée à une autre.

Nous notons graphiquement les multiplicités par un intervalle de la forme `:a..b` où `a` représente la limite inférieure et `b` la limite supérieure. Afin de simplifier dans le cas d'une multiplicité `1..1` on pourra noter simplement `1`.

5.2.3 Concepts

Un **concept** du domaine est défini par son nom, nécessairement unique, le modèle de variabilité [2] qui le représente et la multiplicité qu'il porte.

Définition 5.2.3 (Concept du domaine) :
Nous définissons un concept du domaine $\mathcal{D}$ comme un triplet $\langle id, \mathcal{V}, \mathcal{U} \rangle$ avec id l'identifiant du concept, $\mathcal{V}$ son modèle de variabilité et $\mathcal{U}$ la multiplicité qu'il porte.

Chaque concept doit être unique au sein d'un modèle du domaine. L'unicité des concepts se base sur l'identifiant du concept (deux concepts ne peuvent avoir le même identifiant).

Propriété 5.2.2 (Unicité des concepts) :
Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine,

$\forall\, \mathcal{D}_i \in SC,\ \mathcal{D}_i = \langle id_i, \mathcal{V}_i, \mathcal{U}_i \rangle$ alors $\nexists\, \mathcal{D}_j = \langle id_j, \mathcal{V}_j, \mathcal{U}_j \rangle \in SC$, $i \neq j$, tel que $id_i = id_j$.

1. Pour rappel, la cardinalité d'un ensemble correspond au nombre d'éléments de l'ensemble.
2. Nous formalisons l'expression de la variabilité dans la section suivante.

Exemple 3 : Modèle du domaine et multiplicité des concepts

Notre exemple fil rouge possède 4 concepts uniques : *Immeuble*, *Appartement*, *Pièce* et *Ouverture*.

Un produit d'une LPL pour cet exemple représente un et un seul *Immeuble* : la multiplicité sur ce concept sera donc de `1`. En revanche, il possède au minimum un *Appartement*, une *Pièce* et une *Ouverture* mais n'est théoriquement pas limité quant à leur nombre total : les multiplicités seront alors dans les trois cas `1..*`.

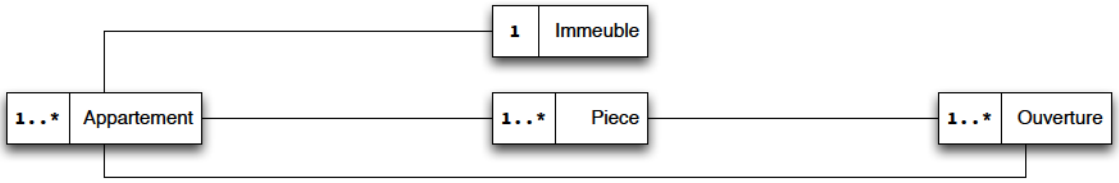


FIGURE 5.2 – Exemple simplifié de modèle du domaine avec les multiplicités des concepts

La Figure 5.2 donne une représentation du modèle du domaine pour l'exemple fil rouge avec les multiplicités sur les concepts.

5.2.4 Associations

Nous définissons une **association** par ses deux extrémités et par un ensemble de fonctions de restriction, que nous présentons dans la suite de ce chapitre (voir section 5.4), qui permettent de modéliser la compatibilité entre les concepts. Une association est par ailleurs définie de manière bidirectionnelle. Une **extrémité d'association** est définie par le concept qu'elle représente et par la multiplicité qu'elle porte.

Définition 5.2.4 (Associations et Extrémités) :
Nous définissons une extrémité d'association $\mathcal{E}$ comme un couple $\langle \mathcal{D}, \mathcal{U} \rangle$ où $\mathcal{D}$ est le concept sur lequel porte l'extrémité de l'association, et $\mathcal{U}$ est la multiplicité portée par l'association pour ce concept, c'est à dire le nombre de fois que le concept pourra être lié à l'autre concept de l'association.

Nous définissons une associations $\mathcal{A}_{ij}$ entre les concepts $\mathcal{D}_i$ et $\mathcal{D}_j$ comme un triplet : $\mathcal{A}_{ij} = \langle \mathcal{E}_i, \mathcal{E}_j, SF \rangle$ où $\mathcal{E}_i$ (resp. $\mathcal{E}_j$) correspond à l'extrémité de l'association pour le concept $\mathcal{D}_i$ (resp. $\mathcal{D}_j$), et SF à l'ensemble $\mathcal{F}_0, ..., \mathcal{F}_n$ des fonctions de restriction de l'association.

Il est important de noter que l'association étant bidirectionnelle, l'ordre dans lequel sont exprimés $\mathcal{E}_i$ et $\mathcal{E}_j$ n'est pas important.

Par ailleurs, afin de simplifier la lecture, nous notons la relation entre une association et les concepts qu'elle relie par la notation suivante : $\mathcal{A}_{ij} \rightsquigarrow \{\mathcal{D}_i, \mathcal{D}_j\}$.

De même que nous avons volontairement limité l'expressivité des multiplicités, nous spécifions des contraintes sur le modèle du domaine afin de garantir certaines propriétés de cohérence et de terminaison sur nos algorithmes exploitant ce formalisme.

De même que pour les concepts, chaque association doit être unique au sein d'un modèle

du domaine. Ainsi, il n'existe pas deux associations reliant les mêmes concepts dans un même modèle. L'unicité des associations se base sur les extrémités de l'association (deux associations ne peuvent avoir des extrémités qui concernent les mêmes concepts).

Propriété 5.2.3 (Unicité des associations) :
Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine.
$\forall (\mathcal{D}_i, \mathcal{D}_j) \in SC^2, \nexists (\mathcal{A}^1_{ij}, \mathcal{A}^2_{ij}) \in SA$ tels que $\mathcal{A}^1_{ij} \rightsquigarrow \{\mathcal{D}_i, \mathcal{D}_j\}$ et $\mathcal{A}^2_{ij} \rightsquigarrow \{\mathcal{D}_i, \mathcal{D}_j\}$.

De plus, le modèle du domaine n'autorise pas des associations réflexives : une association ne peut porter que sur deux concepts distincts.

Propriété 5.2.4 (Association non réflexive) :
Soit $\mathcal{M} = \langle SC, SA \rangle$.
$\forall \mathcal{D} \in SC, \nexists \mathcal{A} \in SA$, telle que $\mathcal{A} \rightsquigarrow \{\mathcal{D}, \mathcal{D}\}$.

Exemple 4 : Modèle du domaine et multiplicité des concepts et des associations

La Figure 5.3 présente le modèle du domaine complet pour notre exemple fil rouge incluant les multiplicités sur les associations.

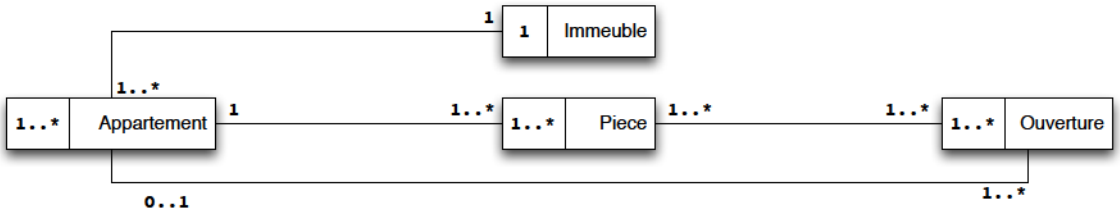


FIGURE 5.3 – Exemple du modèle du domaine pour l'exemple fil rouge

Un *Immeuble* contient forcément un *Appartement* mais peut également en contenir plusieurs et chaque *Appartement* ne peut appartenir qu'à un seul *Immeuble*.

De même, un *Appartement* contient forcément une *Pièce* mais peut également en contenir plusieurs et chaque *Pièce* ne peut appartenir qu'à un seul *Appartement*. Par ailleurs chaque *Pièce* doit forcément avoir au moins une *Ouverture*. Une *Ouverture* est alors réalisée soit entre plusieurs *Pièce* dans le cas d'une *Ouverture* intérieure, soit entre une *Pièce* et l'*Appartement* : il s'agit alors d'une *Ouverture* extérieure.

Enfin, un *Appartement* possède au minimum une *Ouverture* vers l'extérieur. En revanche, toutes les *Ouvertures* ne sont pas des ouvertures extérieures de l'*Appartement*, d'où la multiplicité `0..1`.

5.2.5 Connexité et cohérence du modèle

Le modèle du domaine doit respecter certaines propriétés de cohérence afin de pouvoir être utilisable. Ainsi le modèle doit être connexe : il existe toujours un chemin entre deux concepts quelconques du modèle.

Propriété 5.2.5 (Connexité du modèle du domaine) :
Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine avec SC l'ensemble des concepts et SA l'ensemble des associations.

Soit $(\mathcal{D}_i, \mathcal{D}_j) \in SC^2$ alors il existe toujours un ensemble non vide d'associations $SA_{ij} = \{\mathcal{A}_{ik}, \mathcal{A}_{kl}, ..., \mathcal{A}_{uj}\} \subseteq SA$ formant un chemin entre les concepts tel que $\mathcal{A}_{ik} \rightsquigarrow \{\mathcal{D}_i, \mathcal{D}_k\}$, $\mathcal{A}_{kl} \rightsquigarrow \{\mathcal{D}_k, \mathcal{D}_l\}$, ..., $\mathcal{A}_{uj} \rightsquigarrow \{\mathcal{D}_u, \mathcal{D}_j\}$.

Le modèle doit également être cohérent : les concepts d'un modèle du domaine ne peuvent appartenir qu'à des associations faisant partie du même modèle du domaine.

Propriété 5.2.6 (Cohérence des concepts et associations du modèle) :
Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine avec SC l'ensemble des concepts et SA l'ensemble des associations.

$\forall\, \mathcal{A}_{ij} \in SA$ avec $\mathcal{A}_{ij} \rightsquigarrow \{\mathcal{D}_i, \mathcal{D}_j\}$, alors $\{\mathcal{D}_i, \mathcal{D}_j\} \in SC^2$.

Enfin, les multiplicités doivent également être cohérentes entre-elles :

1. un concept portant une multiplicité avec une limite supérieure non bornée, doit *(i)* soit faire partie d'une association proposant une multiplicité avec une limite supérieure non bornée, *(ii)* soit être lié à un autre concept portant une multiplicité avec une limite supérieure non bornée ;
2. un concept portant une multiplicité avec une limite supérieure égale à 1 doit uniquement faire partie d'associations offrant des multiplicités sur le concept avec une limite supérieure égale à 1 ;
3. un concept portant une multiplicité avec une limite inférieure égale à 0 doit uniquement faire partie d'associations offrant des multiplicités sur le concept avec une limite inférieure égale à 0 [3].

Propriété 5.2.7 (Cohérence des multiplicités du modèle) :
Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine avec SC l'ensemble des concepts et SA l'ensemble des associations.

On dit que les multiplicités sont cohérentes au sein de $\mathcal{M}$ si les règles suivantes sont respectées $\forall\, \mathcal{D}_a \in SC$, tel que $\mathcal{D}_a = \langle id_a, \mathcal{V}_a, \mathcal{U}_a \rangle \in SC$ un concept avec $\mathcal{U}_a$ la multiplicité qui lui est associée et $\mathcal{U}_a = \langle L_a^-, L_a^+ \}$ avec $L_a^- \in \{0, 1\}$ et $L_a^+ \in \{1, *\}$:

1. Si $L_a^+ = *$, alors
   (i) soit $\exists\ \mathcal{A}_{aj} = \langle \mathcal{E}_a, \mathcal{E}_j, SF_{aj} \rangle \in SA$ une association de $\mathcal{M}$ liée à $\mathcal{D}_a$, telle que $\mathcal{E}_a = \langle \mathcal{D}_a, \mathcal{U}_{aja} \rangle$ avec $\mathcal{U}_{aja} = \langle L_{aja}^-, * \rangle$;
   (ii) soit $\exists\ \mathcal{A}_{aj} \in SA$ telle que $\mathcal{A}_{aj} \rightsquigarrow \{\mathcal{D}_a, \mathcal{D}_j\}$ avec $\mathcal{D}_j = \langle id_j, \mathcal{V}_j, \mathcal{U}_j \rangle$ et $\mathcal{U}_j = \langle L_j^-, * \rangle$.
2. Si $L_a^+ = 1$, alors $\forall\ \mathcal{A}_{aj} = \langle \mathcal{E}_a, \mathcal{E}_j, SF_{aj} \rangle \in SA$ une association de $\mathcal{M}$ liée à $\mathcal{D}_a$, on a $\mathcal{E}_a = \langle \mathcal{D}_a, \mathcal{U}_{aja} \rangle$ avec $\mathcal{U}_{aja} = \langle L_{aja}^-, 1 \rangle$.
3. Si $L_a^- = 0$, alors $\forall\ \mathcal{A}_{aj} = \langle \mathcal{E}_a, \mathcal{E}_j, SF_{aj} \rangle \in SA$ une association de $\mathcal{M}$ liée à $\mathcal{D}_a$, on a $\mathcal{E}_a = \langle \mathcal{D}_a, \mathcal{U}_{aja} \rangle$ avec $\mathcal{U}_{aja} = \langle 0, L_{aja}^+ \rangle$.

3. Nous ne présentons pas de concept avec cette multiplicité dans notre exemple.

5.3 Représenter la variabilité par concepts

Comme nous l'avons vu précédemment, nous définissons un modèle du domaine contenant les différents concepts et les associations entre ces concepts. Chaque concept possède son propre modèle de variabilité. Dans le cadre de notre formalisme nous ne considérons que les feature model [4] comme modèle de variabilité. Nous présentons ici la formalisation que nous utilisons des feature models dans le cadre de notre contribution.

5.3.1 Feature Model

Nous représentons la variabilité sous la forme d'un **feature model** (FM), *i.e.* un arbre de features possédant des contraintes internes. Le rôle d'un FM est de modéliser la variabilité d'une famille de produits. Par ailleurs, un FM peut être utilisé au travers d'un processus de configuration dans lequel l'utilisateur sélectionne les features qu'il souhaite et exclut les autres.

Une **feature** est ainsi définie comme un élément de variabilité de la famille de produits ou comme une commonalité [5]. La feature représente une caractéristique du produit.

Chaque feature à l'exception de la racine fait partie d'un **groupe** d'une ou plusieurs features, qui exprime des propriétés sur la façon dont cette ou ces features pourront être sélectionnées lors de la configuration.

Un groupe possède un type qui définit la multiplicité des features du groupe. Il en existe 4 types : **obligatoire**, **optionnel**, **OR** ou **XOR**. Les types *obligatoire* et *optionnel* ne sont utilisés que pour des groupes contenant une unique feature et représentent respectivement les multiplicités `1..1` et `0..1`. Les types OR et XOR sont utilisés pour des groupes de plusieurs features et définissent les multiplicités `1..n` et `1..1`.

Par ailleurs, une feature possède un ou plusieurs groupes de features : on établit ainsi la hiérarchie entre les features. La feature racine n'a par définition, pas de parent, elle n'appartient donc à aucun groupe et elle est considérée comme obligatoire par convention. Cependant, toutes les autres features appartiennent forcément à un groupe défini par leur feature parente. En outre, la sémantique des FM implique que si une feature est sélectionnée lors de la configuration, toutes ses features parentes devront également être sélectionnées.

Définition 5.3.1 (Feature et Groupe) :
Nous définissons une feature $\mathcal{F}$ comme un triplet $\langle id, \mathcal{F}_p, SG \rangle$ avec id l'identifiant de la feature, $\mathcal{F}_p$ sa feature parente ou $\varnothing$ [6] s'il s'agit de la feature racine, et $SG = \{\mathcal{G}_0, ..., \mathcal{G}_n\}$ un ensemble de groupes, SG pouvant être l'ensemble vide. Si SG est l'ensemble vide, on considère alors que la feature est une feuille de l'arbre.

Nous définissons un groupe de feature $\mathcal{G}$ comme un couple $\langle type, SF \rangle$ où $type \in \{Obligatoire, Optionnel, OR, XOR\}$ définit le type de groupe et $SF = \{\mathcal{F}_0, ..., \mathcal{F}_n\}$ est l'ensemble des features du groupe.

La cardinalité de SF est fortement liée au type du groupe :
- si $type \in \{Obligatoire, Optionnel\}$, alors $|SF| = 1$;
- si $type \in \{OR, XOR\}$, alors $|SF| > 1$.

4. Le terme "feature model" est souvent traduit *modèle de caractéristique* ou *modèle de variabilité* en français. La deuxième expression nous semble trop large et la première n'est pas simplement réductible, ni aisément représentative. Nous conserverons donc dans l'ensemble du document le terme *Feature Model* et l'acronyme FM qui lui correspond.

5. Un élément commun à tous les produits.

6. Nous utiliserons dans l'ensemble de notre contribution le symbole $\varnothing$ pour signifier une variable sans valeur.

Exemple 5 : Features et Groupes

Dans le cadre de notre exemple fil rouge, nous avons besoin de représenter les différents capteurs et actionneurs qui peuvent équiper une pièce.

Le sens même de cette phrase nous montre que le FM de *Pièce* possède deux features optionnelles *Capteurs* et *Actionneurs* : ces features sont donc intégrées à des groupes optionnels.

On peut dès lors commencer à créer la hiérarchie suivante de features :

Soit $\mathcal{F}_{piece}$ une feature représentant une *Piece* qui sera la feature racine du FM. Soit $\mathcal{F}_{capteurs}$ et $\mathcal{F}_{actionneurs}$ les features représentant les *Capteurs* et les *Actionneurs*. On a alors : $\mathcal{F}_{piece} = \langle Piece, \varnothing, SG_{piece} \rangle$. Etant la racine, cette feature n'a en effet pas de parent.

La feature $\mathcal{F}_{piece}$ contient deux groupes optionnels pour les *Capteurs* et les *Actionneurs* : $SG_{piece} = \{\mathcal{G}_{actionneurs}, \mathcal{G}_{capteurs}\}$ avec $\mathcal{G}_{actionneurs} = \langle Optionnel, \{\mathcal{F}_{actionneurs}\} \rangle$ et $\mathcal{G}_{capteurs} = \langle Optionnel, \{\mathcal{F}_{capteurs}\} \rangle$.

Les features *Capteurs* et *Actionneurs* sont alors définies en relation avec leur feature parente : $\mathcal{F}_{actionneurs} = \langle Actionneurs, \mathcal{F}_{piece}, SG_{actionneurs} \rangle$ et $\mathcal{F}_{capteurs} = \langle Capteurs, \mathcal{F}_{piece}, SG_{capteurs} \rangle$.

La Figure 5.4 donne une représentation graphique de cette hiérarchie.

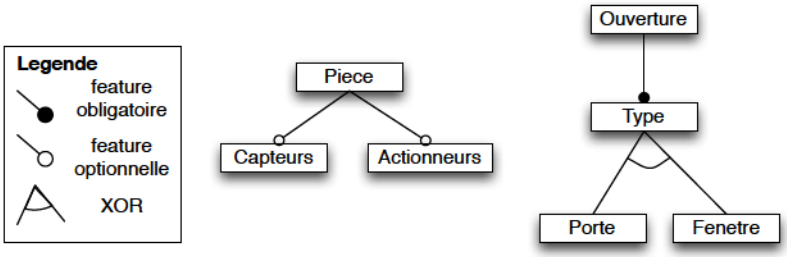


FIGURE 5.4 – Extraits des FM Piece et Ouverture

On constate également dans la Figure 5.4 qu'une *Ouverture* a une feature *Type* pour indiquer s'il s'agit d'une *Fenêtre* ou d'une *Porte*. Cette feature sera donc intégrée à un groupe obligatoire et disposera d'un groupe XOR contenant les features *Fenêtre* et *Porte*.

On a alors la hiérarchie suivante : Soit $\mathcal{F}_{ouverture}$, $\mathcal{F}_{type}$, $\mathcal{F}_{fenetre}$ et $\mathcal{F}_{porte}$ les features représentant une *Ouverture*, un *Type*, une *Fenêtre* et une *Porte*.

On a alors : $\mathcal{F}_{ouverture} = \langle Ouverture, \varnothing, SG_{ouverture} \rangle$.

Cette feature contient un groupe obligatoire contenant la feature *Type* : $SG_{ouverture} = \{\mathcal{G}_{type}\}$ avec $\mathcal{G}_{type} = \langle Obligatoire, \{\mathcal{F}_{type}\} \rangle$.

La feature *Type* est définie par rapport à sa relation à son parent et définit un groupe de features : $\mathcal{F}_{type} = \langle Type, \mathcal{F}_{ouverture}, SG_{type} \rangle$ avec $SG_{type} = \{\mathcal{G}_{xorPorteFenetre}\}$.

On a finalement le groupe exprimant la relation XOR entre *Porte* et *Fenetre* : $\mathcal{G}_{xorPorteFenetre} = \langle XOR, \{\mathcal{F}_{porte}, \mathcal{F}_{fenetre}\} \rangle$ avec $\mathcal{F}_{porte} = \langle Porte, \mathcal{F}_{type}, SG_{porte} \rangle$ et $\mathcal{F}_{fenetre} = \langle Fenetre, \mathcal{F}_{type}, SG_{fenetre} \rangle$.

Une **contrainte** interne au sein d'un FM permet d'exprimer des relations particulières entre certaines features à travers l'arbre, en dehors de la relation hiérarchique. Une contrainte est utilisée lors du processus de configuration afin de vérifier la validité de la sélection réalisée par

l'utilisateur.

Une contrainte s'exprime sous la forme "A implique B" ou "A implique non B" selon qu'il s'agit d'une contrainte d'implication ou d'exclusion, A et B étant des features. Or l'implication logique "A implique B" peut également s'écrire "non A ou B".

Nous définissons ainsi une contrainte comme une clause disjonctive, soit une formule booléenne utilisant uniquement les opérateurs *ou* et *non* et dont les littéraux sont des features. Par ailleurs, les contraintes du FM sont toutes utilisées de manière conjonctive : exprimer "A implique B et C" revient à créer deux contraintes "A implique B" et "A implique C". Les trois opérateurs booléens nécessaires à la construction de n'importe quelle formule booléenne sont donc ici bien présents : le *ou* et le *non* au sein de la contrainte et le *et* entre les contraintes.

Définition 5.3.2 (Contrainte) :
Nous définissons une contrainte $\mathcal{CST}$ comme un couple $\langle ST, SN\rangle$ où ST représente l'ensemble des littéraux *sans* l'opérateur *non* et SN l'ensemble des littéraux *avec* l'opérateur *non*. $\mathcal{CST}$ représente une clause disjonctive, on a donc $ST = \{a_0, a_1, ..., a_n\}$ et $SN = \{b_0, b_1, ..., b_m\}$ alors $\mathcal{CST} \Leftrightarrow a_0 \vee ... \vee a_n \vee \neg b_0 \vee ... \vee \neg b_m$ avec $ST \cap SN = \emptyset$.

Exemple 6 : Contraintes de FM

Une contrainte dans notre exemple fil rouge dit que si un *Capteur* de *Luminosité* est sélectionné dans une *Piece* alors un *Actionneur* pour les *Store* doit être sélectionné. Nous voulons donc exprimer la contrainte : "Luminosite implique Store".

Pour cela, nous devons revenir à une formule logique exprimée sous forme disjonctive. Soit $\mathcal{F}_{CLuminosite}$ et $\mathcal{F}_{AStore}$ les features correspondantes au *Capteur* de *Luminosite* et à l'*Actionneur* de *Store*, alors :

$\mathcal{F}_{CLuminosite} \Rightarrow \mathcal{F}_{AStore} \Leftrightarrow \neg \mathcal{F}_{CLuminosite} \vee \mathcal{F}_{AStore}$

On peut alors créer la contrainte : $\mathcal{CST}_{ls} = \langle ST_{ls}, SN_{ls}\rangle$ avec $ST_{ls} = \{\mathcal{F}_{AStore}\}$ et $SN_{ls} = \{\mathcal{F}_{CLuminosite}\}$. On a donc : $\mathcal{CST}_{ls} \Leftrightarrow \mathcal{F}_{AStore} \vee \neg \mathcal{F}_{CLuminosite}$.

Nous définissons un **feature model** par sa racine, un ensemble de features, un ensemble de groupes et un ensembles de contraintes.

Définition 5.3.3 (Feature Model) :
Nous définissons un feature model $\mathcal{V}$ comme un quadruplet $\langle \mathcal{F}_{root}, SF, SG, SC\rangle$ où $\mathcal{F}_{root}$ est la feature racine du FM,
$SF = \{\mathcal{F}_0, ..., \mathcal{F}_n\}$ représente l'ensemble des features du FM à l'exclusion de la racine,
$SG = \{\mathcal{G}_0, ..., \mathcal{G}_m\}$ représente l'ensemble des groupes de features du FM, et
$SC = \{\mathcal{CST}_0, ..., \mathcal{CST}_n\}$ est un ensemble de contraintes entre les features.

$\forall\, \mathcal{CST}_i \in SC$ avec $\mathcal{CST}_i = \langle ST_i, SN_i\rangle$, $ST_i \cup SN_i \subseteq SF$.

Par souci de simplicité, si $\mathcal{F}_i \in SF$, nous notons directement $\mathcal{F}_i \in \mathcal{V}$. Nous notons la relation entre un FM $\mathcal{V}$ et le concept $\mathcal{D}$ dont il représente la variabilité : $\mathcal{V} \rightsquigarrow \mathcal{D}$.

Nous définissons par ailleurs l'absence de cycle dans la hiérarchie de features par la propriété suivante : $\forall\, \mathcal{F}_i \in SF, \exists\, !\, \mathcal{G}_i = \langle type_i, SF_i\rangle \in SG$ tel que $\mathcal{F}_i \in SF_i$; de plus, $\nexists \mathcal{G}_{root} = \langle type_{root}, SF_{root}\rangle \in SG$ tel que $\mathcal{F}_{root} \in SF_{root}$.

Un FM peut être représenté sous la forme d'une formule booléenne conjonctive (CNF) dans laquelle chaque feature est représentée par un littéral [Mendonca 09]. Nous définissons donc la *cohérence* d'un FM par la satisfiabilité de sa formule pour n'importe quel littéral interprété comme vrai. Dit autrement, un FM est cohérent s'il existe au moins une solution (modèle logique ou *configuration*) à la formule associée à ce FM et chaque feature peut être incluse dans au moins une configuration [7].

Propriété 5.3.1 (Cohérence d'un FM) :
Soit $\mathcal{V} = \langle \mathcal{F}_{root}, SF, SG, SC \rangle$ un FM. Il existe alors une formule CNF Γ dont les littéraux sont l'ensemble des features présentes dans $SF \cup \{\mathcal{F}_{root}\}$ et qui définit formellement la sémantique du FM représentée par ses groupes et ses contraintes.

$\mathcal{V}$ est *cohérent* si $\forall \mathcal{F}_i \in SF \cup \{\mathcal{F}_{root}\}$, il existe un modèle de Γ I_i tel que si $I_i(\mathcal{F}_i)$ est vrai alors Γ est satisfiable.

Sauf mention explicite, nous ne considérons dans la suite du document que des FM cohérents selon cette propriété.

Nous utilisons un formalisme graphique inspiré du formalisme défini par Kang *et al.* et utilisé dans de nombreux travaux pour représenter les FM [Kang 90, Czarnecki 02]. L'arbre est graphiquement représenté de manière verticale avec la racine en haut. Les types de groupes sont représentés directement sur les features pour les groupes unitaires : un *disque noir* indique une feature dans un groupe obligatoire et un *cercle noir vide* indique une feature dans un groupe optionnel. Pour les groupes possédant plusieurs features, les types sont directement indiqués par un arc de cercle : un *arc plein noir* indique un groupe OR, alors qu'un *arc vide* indique un groupe XOR. Enfin, les contraintes sont précisées de manière textuelle sous ou à côté du FM. Afin de simplifier la lecture, nous écrivons directement les implications ($\Rightarrow$) et équivalences ($\Leftrightarrow$) dans la représentation graphique.

Exemple 7 : Feature models de l'exemple fil rouge

Nous présentons dans le cadre de notre exemple fil rouge une variabilité simplifiée afin de conserver une certaine lisibilité dans nos exemples.

La Figure 5.5 montre les quatre FM associés à nos quatre concepts de l'exemple représentés dans le modèle du domaine (voir Figure 5.3).

L'*Immeuble* possède une variabilité élémentaire : il peut posséder un raccordement à la *Fibre optique* et un *Chauffage central*, celui-ci pouvant être alimenté au *Fioul* ou au *Gaz*.

La variabilité d'un *Appartement* est plus complexe. Ainsi, un appartement peut disposer d'un raccordement à la *fibre optique*, un *Ordinateur Central* et doit disposer d'un *Chauffage*. L'*ordinateur central* peut gérer soit la *Temperature*, la *Luminosité* ou encore la *Securité*. Le type de *Chauffage* peut être *Fioul*, *Gaz* ou *Electrique*. Enfin, le *Chauffage* peut être *Centralisé* ou non, et s'il l'est peut fonctionner par *Plancher* chauffant ou par *Radiateur*. Par ailleurs, si l'*ordinateur central* est en charge de la *Sécurité* alors l'appartement doit être raccordé à la *Fibre optique* ; de même, si l'*ordinateur* est en charge de la gestion de la *Temperature*, l'appartement doit être équipé d'un chauffage *Centralisé*.

Une *Piece* peut posséder des *Actionneurs*, des *Capteurs* ou des *Radiateurs*. Nous définissons trois types de *Capteurs* pour détecter des variations de *Temperature*, de *Luminosite*

7. Il ne contient aucune *feature morte*.

ou de *Presence*. Nous définissons également trois types d'*Actionneurs* pour gérer la fermeture des *Stores*, le réglage du *Thermostat* ou encore la *Sécurité* via le déclenchement d'une *Alarme* ou le déclenchement du *Verrou*. Il est important de noter que les features correspondant à des *Capteurs* ou des *Actionneurs* sont préfixées respectivement par *C* ou *A* (par exemple, *CLuminosite* ou *AStore*).

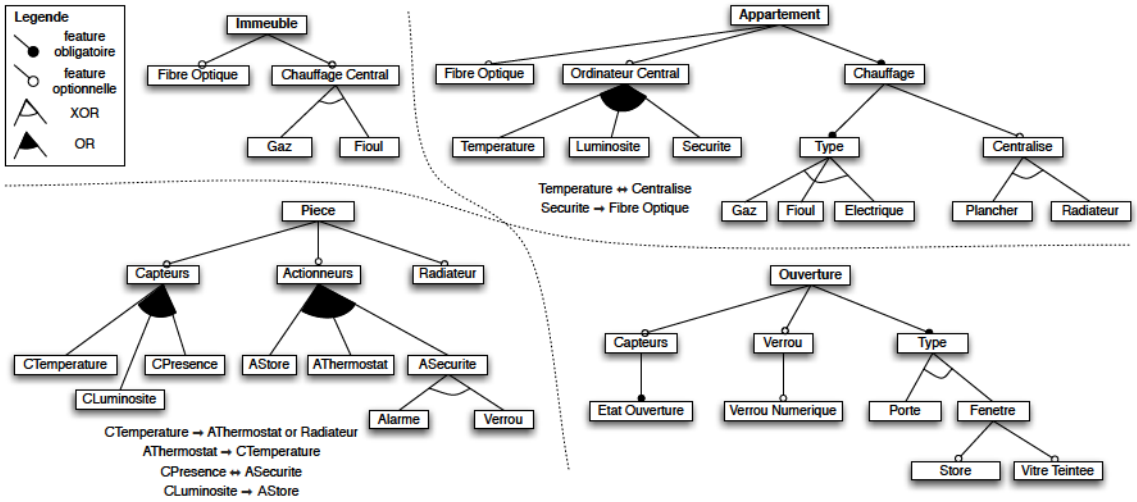


FIGURE 5.5 – Feature Models pour l'exemple fil rouge

Enfin, une *Ouverture* peut être de deux *Type* : il peut s'agir d'une *Fenetre* ou d'une *Porte*. S'il s'agit d'une *Fenetre*, elle pourra avoir des *Stores* ainsi qu'une *Vitre Teintee*. Chaque *Ouverture* peut disposer d'un *Verrou*, qui peut être *Numerique*. Une *Ouverture* peut également disposer d'un *Capteur* qui sera forcément un capteur permettant de déterminer l'*Etat d'Ouverture* de la *Fenetre* ou de la *Porte*.

5.3.2 Configuration de FM

Un feature model est utilisable au travers d'un processus de configuration durant lequel un utilisateur choisira explicitement les features qu'il souhaite pour le produit qu'il configure. Nous définissons ici plusieurs propriétés liées à la configuration des FM.

En premier lieu, nous définissons un **état de configuration** comme une représentation d'un FM, dans lequel certaines features sont sélectionnées et d'autres exclues.

Définition 5.3.4 (Etat de configuration) :
Soit $\mathcal{V} = \langle \mathcal{F}_{root}, SF, SG, SC \rangle$ un feature model.

Nous définissons un état de configuration $\mathcal{S}$ comme un triplet $\langle \mathcal{V}, SS, SE \rangle$ avec $SS = \{\mathcal{F}_i, ..., \mathcal{F}_j\}$ l'ensemble des features sélectionnées et $SE = \{\mathcal{F}_m, ..., \mathcal{F}_n\}$ l'ensemble des features exclues, tel que
$SS \cup SE \subseteq SF \cup \mathcal{F}_{root}$.

Exemple 8 : Etat de configuration

Un état de configuration pour l'*Immeuble* est défini dans le Listing 5.1.

Listing 5.1 – Etat de configuration

```
selections : Immeuble, Chauffage Central

exclusions : Fibre Optique
```

Un état de configuration peut être partiel ou complet : on dit qu'il est complet si toutes les features du FM sont soit sélectionnées soit exclues.

Propriété 5.3.2 (Etat de configuration complet) :
Soit $\mathcal{V} = \langle \mathcal{F}_{root}, SF, SG, SC \rangle$ un feature model. Soit $\mathcal{S} = \langle \mathcal{V}, SS, SE \rangle$ un état de configuration qui lui est associé.

$\mathcal{S}$ est *complet* si et seulement si $SS \cup SE = SF \cup \mathcal{F}_{root}$.

Exemple 9 : Etat de configuration complet

Un état de configuration complet pour l'*Immeuble* est présenté dans le Listing 5.2 : toutes les features du FM *Immeuble* sont dans un état sélectionné ou exclu.

Listing 5.2 – Etat de configuration

```
selections : Immeuble, Chauffage Central, Gaz

exclusions : Fibre Optique, Fioul
```

Un état de configuration est *cohérent* relativement à un FM, si le FM est cohérent et l'ensemble de ses contraintes (type des groupes et contraintes internes) sont respectées. On peut expliciter les règles ainsi :

1. aucune feature ne peut être à la fois sélectionnée et exclue ;
2. la racine est forcément sélectionnée ;
3. pour toute feature sélectionnée, son parent doit également être sélectionné ;
4. une feature appartenant à un groupe obligatoire est forcément sélectionnée si son parent est sélectionné ;
5. deux features appartenant à un groupe XOR ne peuvent être sélectionnées en même temps ;
6. si une feature sélectionnée possède un groupe OR ou XOR, au moins une feature du groupe doit être sélectionnée ;
7. les contraintes du FM sont respectées.

Propriété 5.3.3 (Etat de configuration cohérent) :
Soit $\mathcal{V} = \langle \mathcal{F}_{root}, SF, SG, SC \rangle$ un feature model cohérent (voir Propriété 5.3.1).

Soit $\mathcal{S} = \langle \mathcal{V}, SS, SE \rangle$ un état de configuration qui lui est associé.

$\mathcal{S}$ est *cohérent* par rapport à la sémantique de $\mathcal{V}$ si toutes les propriétés suivantes sont vérifiées :

1. $SS \cap SE = \emptyset$;

2. $\mathcal{F}_{root} \in SS$;
3. $\forall\, \mathcal{F}_i \in SS \setminus \mathcal{F}_{root}$, avec $\mathcal{F}_i = \langle id_i, \mathcal{F}_p, SG_i \rangle$, alors $\mathcal{F}_p \in SS$;
4. $\forall\, \mathcal{G}_m \in SG$ tel que $\mathcal{G}_m = \langle Obligatoire, \{\mathcal{F}_i\} \rangle$, avec $\mathcal{F}_i = \langle id_i, \mathcal{F}_p, SG_i \rangle$, si $\mathcal{F}_p \in SS$ alors $\mathcal{F}_i \in SS$;
5. $\forall\, \{\mathcal{F}_i, \mathcal{F}_j\} \subseteq SS, \nexists\, \mathcal{G} = \{XOR, SF_g\} \in SG$ tel que $\{\mathcal{F}_i, \mathcal{F}_j\} \subseteq SF_g$;
6. $\forall\, \mathcal{F}_i \in SS \setminus \mathcal{F}_{root}$, avec $\mathcal{F}_i = \langle id_i, \mathcal{F}_p, SG_i \rangle$, $\forall\, \mathcal{G}_i \in SG_i$ tel que $\mathcal{G}_i = \langle type, SF_i \rangle$ avec $type \in \{OR, XOR\}$ alors $SF_i \cap SS \neq \emptyset$;
7. $\forall\, \mathcal{CST}_i \in SC, \mathcal{CST}_i \Leftrightarrow \mathcal{F}_0 \vee ... \vee \mathcal{F}_{n_i}$ alors $\mathcal{CST}_i$ doit être satisfiable selon le modèle logique donné par l'état de configuration.

Nous pouvons alors définir la notion de *configuration d'un FM* par rapport à un état de configuration.

Définition 5.3.5 (Configuration d'un FM) :
Nous définissons la configuration d'un FM comme un état de configuration complet et cohérent. Par abus de langage et afin de simplifier le discours, nous nommons configuration partielle un état de configuration qui est cohérent mais non complet.

Etant donné un FM $\mathcal{V}$, nous notons $[\![\mathcal{V}]\!]$ l'ensemble des configurations complètes et cohérentes de $\mathcal{V}$.

Il est important de noter que la cohérence de l'état de configuration est indépendante de sa complétude : un état de configuration partiel peut être cohérent et réciproquement. De même, un état de configuration partiel peut être incohérent selon la Propriété 5.3.3 sans que la sémantique du FM soit violée : des features peuvent encore être sélectionnées ou exclues pour rétablir la cohérence.

En revanche, un état de configuration peut également violer la sémantique du FM, qu'il soit complet ou non. Nous introduisons donc la notion d'état de configuration *faux* relativement à un FM. Un état de configuration est faux si le FM n'est pas cohérent *ou* si le FM est cohérent et *au moins une* de ces règles est vérifiée :

1. il existe une feature à la fois sélectionnée et exclue ;
2. la racine du FM est exclue ;
3. il existe une feature sélectionnée dont le parent est exclu ;
4. il existe une feature obligatoire exclue alors que son parent est sélectionné ;
5. deux features d'un même groupe XOR sont sélectionnées ;
6. il existe un groupe OR ou XOR dont le parent est sélectionné mais dont tous les membres sont exclus ;
7. une contrainte dont toutes les features sont sélectionnées ou exclues n'est pas respectée.

Propriété 5.3.4 (Etat de configuration faux) :
Soit $\mathcal{V} = \langle \mathcal{F}_{root}, SF, SG, SC \rangle$ un feature model.

Si $\mathcal{V}$ n'est pas cohérent selon la Propriété 5.3.1 alors l'état de configuration est considéré comme faux.

Si $\mathcal{V}$ est cohérent, soit $\mathcal{S} = \langle \mathcal{V}, SS, SE \rangle$ un état de configuration qui lui est associé.

Alors $\mathcal{S}$ est *faux* par rapport à la sémantique de $\mathcal{V}$ si au moins l'une des propriétés suivantes n'est pas vérifiée :

1. $SS \cap SE \neq \emptyset$;
2. $\mathcal{F}_{root} \notin SS$;
3. $\exists\, \mathcal{F}_i \in SS \setminus \mathcal{F}_{root}$ telle que $\mathcal{F}_i = \langle id_i, \mathcal{F}_p, SG_i \rangle$ avec $\mathcal{F}_p \in SE$;
4. $\exists\, \mathcal{G}_i \in SG$ tel que $\mathcal{G}_i = \langle Obligatoire, \{\mathcal{F}_i\} \rangle$ avec $\mathcal{F}_i = \langle id_i, \mathcal{F}_p, SG_i \rangle$ et $\mathcal{F}_i \in SE$ alors que $\mathcal{F}_p \in SS$;
5. $\exists\, \mathcal{G}_i \in SG$ tel que $\mathcal{G}_i = \langle XOR, SF_i \rangle$ avec $SF_i = \{\mathcal{F}_0, ..., \mathcal{F}_n\}$ et $|SF_i \cap SS| > 1$;
6. $\exists\ \mathcal{F}_i \in SS$ tel que $\mathcal{F}_i = \langle id_i, \mathcal{F}_p, SG_i \rangle$, $\exists\ \mathcal{G}_i \in SG_i$ tel que $\mathcal{G}_i = \langle type, SF_i \rangle$ avec $type \in \{XOR, OR\}$ et $SF_i = \{\mathcal{F}_0, ..., \mathcal{F}_n\}$ et $SF_i \cap SE = SF_i$;
7. $\exists\, \mathcal{CST}_i \in SC$ avec $\mathcal{CST}_i = \langle ST, SN \rangle$ telle que $ST \cup SN \subseteq SS \cup SE$ et $\mathcal{CST}_i$ n'est pas satisfiable par le modèle logique donné par l'état de configuration.

Sauf mention explicite tous les états de configuration que nous mentionnons dans le reste du document sont considérés comme cohérents.

Exemple 10 : Etat de configuration complet, partiel et faux

L'exemple suivant pour le FM *Piece* décrit dans la Figure 5.5 est *complet et cohérent* :

Listing 5.3 – Etat de configuration complet et cohérent

```
selections  : Piece, Capteurs, CTemperature, Actionneurs, AThermostat, Radiateur

exclusions  : CPresence, CLuminosite, AStore,  ASecurite, Alarme, Verrou
```

Toutes les features sont en effet présentes, soit dans le set des éléments sélectionnés soit dans le set des éléments exclus.

Un état de configuration *partiel et cohérent* pour le même FM serait le suivant :

Listing 5.4 – Etat de configuration partiel et cohérent

```
selections  : Piece, Capteurs, CPresence, Actionneurs, ASecurite

exclusions  : CLuminosite, AStore, AThermostat
```

On constate qu'il manque bien des features, en particulier aucun choix n'a été fait entre *Alarme* et *Verrou*.

Enfin, un état de configuration *faux* pour le même FM serait le suivant :

Listing 5.5 – Etat de configuration faux

```
selections  : Piece, Alarme, CTemperature

exclusions  : CLuminosite, AStore, Radiateur, AThermostat, CPresence, ASecurite, Verrou
```

On constate ici plusieurs incohérences par rapport à la sémantique du FM. Tout d'abord, *Alarme* ne peut être sélectionnée et *ASecurite* exclue cette dernière étant le parent de *Alarme*. Par ailleurs une contrainte spécifie que si *CTemperature* est sélectionnée alors *Radiateur* ou *AThermostat* doit être sélectionné, or les deux sont ici exclus. L'état de configuration est donc faux.

Propriété 5.3.5 (Etat de configuration et opérations ensemblistes) :
On définit les opérations ensemblistes entre deux états de configuration se référant au même FM, en effectuant indépendamment les opérations entre les ensembles contenant les éléments sélectionnés et les ensembles contenant les éléments exclus.

Ainsi, en considérant $\mathcal{S}_i = \langle \mathcal{V}, SS_i, SE_i \rangle$ et $\mathcal{S}_j = \langle \mathcal{V}, SS_j, SE_j \rangle$, on a :

— $\mathcal{S}_i \; op \; \mathcal{S}_j \Leftrightarrow \langle \mathcal{V}, SS_i \; op \; SS_j, SE_i \; op \; SE_j \rangle$ avec $op \in \{\cup, \cap\}$
— et $\mathcal{S}_i \; op \; \mathcal{S}_j \Leftrightarrow (SS_i \; op \; SS_j) \wedge (SE_i \; op \; SE_j)$ avec $op \in \{\subset, \supset, \subseteq, \supseteq, =\}$.

Nous définissons par extension l'ensemble des configurations atteignables à partir d'un état de configuration.

Propriété 5.3.6 (Ensemble de configurations atteignables à partir d'un état de configuration) :
Soit $\mathcal{V}$ un FM. Soit $\mathcal{S} = \langle \mathcal{V}, SS, SE \rangle$ un état de configuration cohérent.

Nous définissons $[\![\mathcal{S}]\!]$ l'ensemble des configurations atteignables à partir de $\mathcal{S}$ avec :
$[\![\mathcal{S}]\!] = \{\mathcal{S}' | \forall \; \mathcal{S}' = \langle \mathcal{V}, SS', SE' \rangle$ avec $\mathcal{S}' \subseteq \mathcal{S}$ et $\mathcal{S}'$ respecte la Propriété 5.3.2 et la Propriété 5.3.3$\}$.

Nous avons nécessairement $[\![\mathcal{S}]\!] \subseteq [\![\mathcal{V}]\!]$.

Si $\mathcal{S}$ est complet alors $[\![\mathcal{S}]\!] = \{\mathcal{S}\}$.

La complexité d'un FM peut être définie par de nombreuses métriques comme son nombre de features, de contraintes ou encore le ratio entre le nombre de features et de contraintes [Bagheri 11]. Dans notre formalisme, nous sommes principalement intéressés de connaître le nombre de configurations réalisables à partir d'un FM. Ainsi nous définissons la *complexité* d'un FM, comme le nombre de configurations atteignables par le FM. Par extension, nous définissons la *complexité* d'un état de configuration comme le nombre de configurations atteignables à partir de l'état de configuration courant.

Propriété 5.3.7 (Complexité d'un FM et d'un état de configuration) :
Soit $\mathcal{V}$ un FM.

Nous notons la complexité d'un FM $\theta(\mathcal{V})$. Nous définissons cette complexité $\theta(\mathcal{V}) = | \; [\![\mathcal{V}]\!] \; |$.

Soit $\mathcal{S} = \langle \mathcal{V}, SS, SE \rangle$ un état de configuration de $\mathcal{V}$. Nous notons la complexité d'un état de configuration $\theta(\mathcal{S})$. Nous définissons la complexité d'un état de configuration $\theta(\mathcal{S}) = | \; [\![\mathcal{S}]\!] \; |$.

5.4 Gérer les contraintes entre les modèles de variabilité

Le modèle du domaine nous permet de représenter la variabilité complexe d'un domaine en la considérant concept par concept. Cependant, les différents modèles de variabilité peuvent être interdépendants : les produits exprimés dans chaque famille de produits ne sont pas nécessairement tous compatibles relativement aux associations. Ainsi, il est nécessaire de représenter des contraintes entre les FM des différents concepts. Nous proposons un mécanisme d'expression de ces contraintes basé sur l'état de configuration du FM et sur une action à réaliser sur un état de configuration d'un FM dépendant.

5.4.1 A propos d'actions de configuration sur les FM

Si différents types d'actions sont disponibles dans notre formalisme d'action présenté au chapitre suivant (voir section 6.5), nous ne considérons dans nos contraintes que des actions à appliquer sur les FM. Ainsi 3 types d'actions sont possibles :

1. sélectionner une feature dans un FM ;
2. exclure une feature d'un FM ;
3. ajouter une nouvelle contrainte dans un FM.

Ces actions ont toutes pour caractéristique de restreindre l'ensemble des produits disponibles à partir du FM : sélectionner ou exclure une feature implique de ne conserver que les produits incluant ou excluant la feature en question, et ajouter une nouvelle contrainte permet de discriminer davantage de features.

Définition 5.4.1 (Action de configuration sur les FM) :
Nous définissons une action sur un FM $\mathcal{A}[fm]$ comme un triplet $\langle type, \mathcal{V}, element\rangle$ où $type \in \{Selectionne, Exclut, AjouteContrainte\}$ représente le type de l'action, $\mathcal{V}$ le FM sur lequel doit s'effectuer l'action et $element$ représente une feature à sélectionner ou exclure dans le cas des deux premiers types, ou une contrainte à ajouter dans le dernier type.

Nous définissons pour les actions sur les FM une opération $apply$ qui prend deux arguments : l'action à effectuer et l'état de configuration qui sera modifié à la suite de l'action. Cette opération retourne un état de configuration modifié.

Soit $\mathcal{V} = \langle \mathcal{F}_{root}, SF, SG, SC\rangle$ un FM, $\mathcal{F}_i \in SF$ une feature du FM, et
— $\mathcal{A}[fm]_s = \langle Selectionne, \mathcal{V}, \mathcal{F}_i\rangle$,
— $\mathcal{A}[fm]_e = \langle Exclut, \mathcal{V}, \mathcal{F}_i\rangle$ et
— $\mathcal{A}[fm]_a = \langle AjouteContrainte, \mathcal{V}, \mathcal{CST}\rangle$,
trois actions s'appliquant sur $\mathcal{V}$ où $\mathcal{CST}$ est une contrainte quelconque.

Soit $\mathcal{S} = \langle \mathcal{V}, SS, SE\rangle$ un état de configuration associé à $\mathcal{V}$.

On a alors :
— $apply(\mathcal{A}[fm]_s, \mathcal{S}) \rightarrow \mathcal{S}' = \langle \mathcal{V}, SS', SE\rangle$ tel que $SS' = SS \cup \{\mathcal{F}_i\}$;
— $apply(\mathcal{A}[fm]_e, \mathcal{S}) \rightarrow \mathcal{S}' = \langle \mathcal{V}, SS, SE'\rangle$ tel que $SE' = SE \cup \{\mathcal{F}_i\}$;
— $apply(\mathcal{A}[fm]_a, \mathcal{S}) \rightarrow \mathcal{S}' = \langle \mathcal{V}', SS, SE\rangle$ avec $\mathcal{V}' = \langle \mathcal{F}_{root}, SF, SG, SC'\rangle$ tel que $SC' = SC \cup \{\mathcal{CST}\}$;

On peut noter que selon l'appartenance de $\mathcal{F}_i$ à SS ou SE et selon l'action appliquée, l'état de configuration résultant $\mathcal{S}'$ peut être faux ou exactement identique à $\mathcal{S}$, les opérations sur SS et SE étant ensemblistes.

Soit $SA = \{\mathcal{A}[fm]_0, ..., \mathcal{A}[fm]_n\}$ un ensemble d'actions s'appliquant sur le même FM $\mathcal{V}$. Nous définissons par extension l'opération $apply$ sur la liste d'actions SA comme suit :
$apply(SA, \mathcal{S}) = apply(A[fm]_n, ...apply(A[fm]_0, \mathcal{S})...)$.

Par ailleurs, si $SA = \emptyset$, alors $apply(SA, \mathcal{S}) \rightarrow \mathcal{S}$. Notons que nous travaillons sur des ensembles d'actions, l'ordre des actions n'ayant aucune importance puisque nous procédons par union ensembliste.

Par définition, l'application d'une action de configuration prend en paramètre un état de configuration et retourne un état de configuration. L'état de configuration résultant a nécessairement une *complexité égale ou inférieure* (voir Propriété 5.3.7) à l'état de configuration donné en entrée. Cela est vrai car l'état de configuration résultant restreint les futures actions possibles menant à un état de configuration complet et cohérent (voir Propriété 5.3.2 et Propriété 5.3.3).

Propriété 5.4.1 (Action et complexité des états de configuration) :
Soit $\mathcal{S}$ un état de configuration et $\mathcal{A}[fm]$ une action.
En considérant $apply(\mathcal{A}[fm], \mathcal{S}) \rightarrow \mathcal{S}'$, on a alors : $\theta(\mathcal{S}) \geq \theta(\mathcal{S}')$.

On définit la notion d'actions contradictoires comme un ensemble d'actions menant à un état de configuration *faux*.

Propriété 5.4.2 (Actions contradictoires) :
Soit $\mathcal{S}$ un état de configuration valide et $\mathcal{A}[fm]$ une action. Nous définissons $\mathcal{A}[fm]_i$ une action contradictoire à $\mathcal{A}[fm]$, telle que :
$apply(\mathcal{A}[fm]_i, apply(\mathcal{A}[fm], \mathcal{S})) \rightarrow \mathcal{S}'$ avec $\mathcal{S}'$ un état de configuration *faux* (voir Propriété 5.3.4).
Par extension un ensemble d'actions est contradictoire si la propriété est vérifiée sur l'application de l'ensemble des actions.

Ainsi l'action contradictoire d'une sélection est une exclusion et l'action contradictoire d'un ajout de contrainte est un ajout d'une contrainte menant à une formule contradictoire.

Propriété 5.4.3 (Construction d'actions contradictoires) :
Soit $\mathcal{V} = \langle \mathcal{F}_{root}, SF, SG, SC \rangle$ un FM, $\mathcal{F}_i \in SF$ une feature du FM, et
- $\mathcal{A}[fm]_s = \langle Selectionne, \mathcal{V}, \mathcal{F}_i \rangle$,
- $\mathcal{A}[fm]_e = \langle Exclut, \mathcal{V}, \mathcal{F}_i \rangle$ et
- $\mathcal{A}[fm]_a = \langle AjouteContrainte, \mathcal{V}, \mathcal{CST} \rangle$.

On identifie alors les actions contradictoires suivantes :
- $\mathcal{A}[fm]_e$ est l'action contradictoire de $\mathcal{A}[fm]_s$;
- réciproquement $\mathcal{A}[fm]_s$ est l'action contradictoire de $\mathcal{A}[fm]_e$;
- soit $\mathcal{A}[fm]_{ai} = \langle AjouteContrainte, \mathcal{V}, \mathcal{CST}_i \rangle$ une action contradictoire de $\mathcal{A}[fm]_a$ avec $\mathcal{CST}_i$ une contrainte telle que la conjonction de $\mathcal{CST}$ et $\mathcal{CST}_i$ donne lieu à une formule logique contradictoire.

Propriété 5.4.4 (Composition d'actions, idempotence et ordre) :
Soit $\mathcal{S}$ un état de configuration du FM $\mathcal{V}$ et $\mathcal{A}[fm]_0$ $\mathcal{A}[fm]_1$ deux actions quelconques s'appliquant sur $\mathcal{V}$.
On peut définir la composition d'action, $\mathcal{A}[fm]_0 \circ \mathcal{A}[fm]_1$ telle que :
$apply(\mathcal{A}[fm]_0 \circ \mathcal{A}[fm]_1, \mathcal{S}) = apply(SA, \mathcal{S})$ avec $SA = \{\mathcal{A}[fm]_0, \mathcal{A}[fm]_1\}$.
L'ordre de la composition des actions sur les FM n'a pas d'importance : les actions ne sont jamais interdépendantes, le résultat sur l'état de configuration sera donc toujours le même quel que soit l'ordre.
Cela est vrai car on peut considérer le FM comme une formule CNF et les actions comme un ensemble de valeurs pour les littéraux de cette formule : quel que soit l'ordre dans lequel les littéraux sont donnés la vérification de la formule reste la même [Mendonca 09].
De fait, $apply(\mathcal{A}[fm]_0 \circ \mathcal{A}[fm]_1, \mathcal{S}) \Leftrightarrow apply(\mathcal{A}[fm]_1 \circ \mathcal{A}[fm]_0, \mathcal{S})$.
Par ailleurs, chaque action sur les FM est idempotente : l'application de la même action plusieurs fois ne changera pas l'état de configuration ni le FM.
Ainsi, $apply(\mathcal{A}[fm]_0 \circ \mathcal{A}[fm]_0, \mathcal{S}) \Leftrightarrow apply(\mathcal{A}[fm]_0, \mathcal{S})$.

Enfin, nous pouvons construire pour chaque FM une liste bornée d'actions réalisables dans ce FM.

Propriété 5.4.5 (Finitude de la liste des actions réalisables dans un FM) :
Soit $\mathcal{V} = \langle \mathcal{F}_{root}, SF, SG, SC \rangle$ un FM quelconque. Soit $n = |SF|$ le nombre total de features du FM. Soit SC_{all} l'ensemble des contraintes possibles du FM.

Nous pouvons construire la liste SA des actions définies pour ce FM, $SA = S_{selectionne} \cup S_{exclu} \cup S_{contrainte}$ avec :

— $S_{selectionne} = \bigcup \mathcal{A}[fm]_i$ avec $\mathcal{A}[fm]_i = \langle Selectionne, \mathcal{V}, \mathcal{F}_i \rangle, \forall \mathcal{F}_i \in SF$;
— $S_{exclu} = \bigcup \mathcal{A}[fm]_j$ avec $\mathcal{A}[fm]_j = \langle Exclut, \mathcal{V}, \mathcal{F}_j \rangle, \forall \mathcal{F}_j \in SF$;
— $S_{contrainte} = \bigcup \mathcal{A}[fm]_k$ avec $\mathcal{A}[fm]_k = \langle AjouteContrainte, \mathcal{V}, \mathcal{CST}_k \rangle, \forall \mathcal{CST}_k \in SC_{all}$.

La liste SA est toujours bornée et cette borne maximale est dépendante du nombre de features du FM. Chaque contrainte peut s'exprimer sous la forme d'une formule conjonctive contenant au maximum l'ensemble des features. Le nombre de contraintes possibles est donc borné par le nombre de sous-ensembles que l'on peut obtenir à partir de deux fois la liste des features du FM : chaque feature peut en effet être considérée dans la formule conjonctive sous sa forme inverse ($\neg a$). Dans notre cas la taille de SC_{all} est donc bornée par 2^{2n} avec n le nombre de features du FM.

Il existe donc au plus autant d'actions *AjouteContrainte* que la taille maximale de SC_{all}. En outre, le nombre d'actions *Selectionne* et *Exclut* est intrinsèquement dépendant du nombre de features.

SA est donc ***finie*** et bornée : $|SA| < 2n + 2^{2n}$ où $n = |SF|$.

5.4.2 Définition des règles et fonctions

De la même façon que des contraintes sont utilisées de manière interne au FM pour exprimer des implications ou des exclusions entre des features, nous avons besoin d'exprimer des contraintes *entre* les FM.

Pour cela, nous nous inspirons du modèle de contraintes de Dhungana *et al.* utilisé dans la solution *Invar* [Dhungana 13].

Nous définissons des **règles de restriction** qui, à partir de l'état de configuration d'un FM, vont déterminer une **action** à appliquer sur un autre FM.

Définition 5.4.2 (Règle de restriction) :
Nous définissons une règle de restriction $\mathcal{R}_{source \to cible}$ orientée du FM $source$ vers le FM $cible$ comme un couple $\langle \mathcal{S}, \mathcal{A}[fm] \rangle$ avec $\mathcal{S}$ un état de configuration minimum à atteindre relatif à $\mathcal{V}_{source}$ et $\mathcal{A}[fm]$ une action à appliquer sur un état de configuration de $\mathcal{V}_{cible}$.

Une règle de restriction n'est applicable que si l'état de configuration qu'elle définit est atteint par un état de configuration en cours.

Propriété 5.4.6 (Applicabilité d'une règle de restriction) :
Soit $\mathcal{R}_{source \to cible} = \langle \mathcal{S}, \mathcal{A}[fm] \rangle$ une règle de restriction.

Soit $\mathcal{S} = \langle \mathcal{V}_{source}, SS_{source}, SE_{source} \rangle$ et $\mathcal{A}[fm] = \langle type, \mathcal{V}_{cible}, element \rangle$.

La propriété d'applicabilité spécifie que la règle $\mathcal{R}_{source \to cible}$ s'applique à $\mathcal{S}_{source}$ si et

seulement si $\mathcal{S} \subseteq \mathcal{S}_{source}$.

Nous pouvons ainsi définir l'opération d'application d'une règle de restriction qui se base sur cette propriété d'applicabilité.

Propriété 5.4.7 (Application de règle de restriction) :
Nous définissons l'opération $applyRule$ qui prend en argument la règle de restriction à appliquer, l'état de configuration relatif au concept source de la règle, et l'état de configuration relatif au concept cible de la règle. L'opération retourne un état de configuration cible modifié par l'action de la règle de restriction si et seulement si la propriété d'applicabilité est respectée (voir Propriété 5.4.6).

Soit $\mathcal{R}_{source \to cible} = \langle \mathcal{S}, \mathcal{A}[fm] \rangle$ une règle de restriction.

Soit $\mathcal{S} = \langle \mathcal{V}_{source}, SS_{source}, SE_{source} \rangle$ et $\mathcal{A}[fm] = \langle type, \mathcal{V}_{cible}, element \rangle$.

Soit $\mathcal{S}_{source}$ et $\mathcal{S}_{cible}$ des états de configuration relatifs à $\mathcal{V}_{source}$ et à $\mathcal{V}_{cible}$.

$applyRule(\mathcal{R}_{source \to cible}, \mathcal{S}_{source}, \mathcal{S}_{cible}) \rightarrow apply(\mathcal{A}[fm], \mathcal{S}_{cible})$ si et seulement si $\mathcal{S} \subseteq \mathcal{S}_{source}$, sinon $applyRule(\mathcal{R}_{source \to cible}, \mathcal{S}_{source}, \mathcal{S}_{cible}) \rightarrow \mathcal{S}_{cible}$.

Par extension nous pouvons définir également la même fonction pour un ensemble de règles de restriction : soit $SR = \{\mathcal{R}_0, ..., \mathcal{R}_n\}$ avec $\forall\, \mathcal{R}_i \in SR$, $\mathcal{R}_i = \langle \mathcal{S}_i, \mathcal{A}[fm]_i \rangle$.

On a alors $applyRule(SR, \mathcal{S}_s, \mathcal{S}_c) \Rightarrow apply(\mathcal{A}[fm]_0 \circ ... \circ \mathcal{A}[fm]_n, \mathcal{S}_c)$ pour tous les $\mathcal{S}_s$ avec $\mathcal{S}_i \subseteq \mathcal{S}_s$, $0 \leqslant i \leqslant n$. Et $\mathcal{S}_s$ reste inchangé à la suite de cette opération. Ainsi, l'application d'un ensemble de règles de restriction ne dépend pas de l'ordre d'application des règles.

Afin de simplifier la notation des règles de restriction, nous utilisons une notation textuelle explicite qui *(i)* définit les concepts (et donc les FM) sources et cibles, *(ii)* présente en partie gauche l'état de configuration relatif au FM source à atteindre et *(iii)* présente en partie droite l'action à effectuer sur l'état de configuration relatif au FM cible.

Exemple 11 : Règle de restriction entre Piece et Ouverture

Une règle de restriction peut être créée entre les FM *Pièce* et *Ouverture* représentés dans la Figure 5.5 afin de représenter l'exigence suivante : "si un *Actionneur Verrou* est sélectionné dans *Piece* un *Verrou* doit être sélectionné dans *Ouverture*".

Cette règle va définir un état minimum à atteindre sur le FM *Piece* contenant uniquement la feature *Verrou* dans l'ensemble des features sélectionnées et par une action de sélection sur le FM *Ouverture* prenant comme argument la feature *Verrou* de ce dernier FM.

On notera la règle de la façon suivante :

Listing 5.6 – Exemple de règle de restriction entre Piece et Ouverture

```
regle piece_ouverture_verrou

source Piece
cible Ouverture

selections [Verrou], exclusions [] => SELECTIONNE Verrou
```

Il est cependant possible d'exprimer des contraintes plus complexes, notamment en utilisant l'action *AjouteContrainte*. On peut par exemple représenter l'exigence suivante : "si un *Actionneur Store* est sélectionné alors une *Ouverture* est soit une *Porte* soit une *Fenetre* avec *Store*".

Cette exigence s'exprime en créant un ajout de contrainte dans le FM *Ouverture* spécifiant que soit *Porte* est sélectionné, soit *Store* doit être sélectionné, lorsqu'une *Piece* contient un *Actionneur* de *Store*.

Listing 5.7 – Règle de restriction complexe entre Piece et Ouverture

```
regle piece_ouverture_store

source Piece
cible Ouverture

selections [AStore], exclusions [] => AjouteContrainte Store or Porte
```

Il est également possible de créer des règles de restriction à partir d'un état de configurations plus complexe. Par exemple entre *Piece* et *Appartement* on peut vouloir exprimer l'exigence suivante : "si un *Actionneur* sur le *Thermostat* est sélectionné dans *Piece* sans *Radiateur* alors l' *Appartement* dispose d'un chauffage par le *Plancher*".

Cette contrainte sera réalisée de la façon suivante :

Listing 5.8 – Règle de restriction entre Piece et Appartement

```
regle piece_appartement_plancher

source Piece
cible Appartement

selections [AThermostat], exclusions [Radiateur] => SELECTIONNE Plancher
```

Nous regroupons un ensemble de règles par **fonction de restriction**. Les fonctions sont également orientées d'un concept source vers un concept cible, toutes les règles contenues dans une fonction doivent avoir la même orientation.

Par ailleurs, l'ensemble des règles de restriction au sein d'une fonction de restriction est traité en conjonction : l'application de la fonction de restriction consiste à appliquer l'ensemble des règles.

Définition 5.4.3 (Fonction de restriction) :
Nous définissons une fonction de restriction $\mathcal{RF}$ comme un triplet $\langle \mathcal{D}_{source}, \mathcal{D}_{cible}, SR \rangle$ avec $\mathcal{D}_{source}$ son concept source, $\mathcal{D}_{cible}$ son concept cible et $SR = \{\mathcal{R}_0, ..., \mathcal{R}_n\}$ l'ensemble des règles qu'elle contient.

$\forall\, \mathcal{R}_i \in SR, \mathcal{R}_i = \langle \mathcal{S}_i, \mathcal{A}[fm]_i \rangle$, on a $\mathcal{S}_i = \langle \mathcal{V}_{source}, SS_i, SE_i \rangle$ et $\mathcal{A}[fm]_i$ s'applique à $\mathcal{V}_{cible}$, avec $\mathcal{V}_{source} \rightsquigarrow \mathcal{D}_{source}$ et $\mathcal{V}_{cible} \rightsquigarrow \mathcal{D}_{cible}$.

On note $\mathcal{RF}_{i \to j}$ pour exprimer une fonction orientée du concept i vers le concept j.

Bien qu'une règle ne permette d'exprimer qu'une unique action, l'ensemble des règles associées à la fonction de restriction va permettre d'exprimer des compositions d'actions pour un état donné.

Nous définissons l'application d'une fonction de restriction sur un couple d'états de configuration correspondants aux états respectifs des FM des concepts sources et cibles, comme l'application de chacune des règles de restriction de la fonction sur le couple.

Propriété 5.4.8 (Application d'une fonction de restriction) :
Soit $\mathcal{S}_{source}$ un état de configuration associé au FM de $\mathcal{D}_{source}$ et $\mathcal{S}_{cible}$ un état de configuration

associé au FM de $\mathcal{D}_{cible}$. Nous définissons l'opération $applyRF$ qui prend en argument la fonction de restriction $\mathcal{RF}$ à appliquer et les états de configuration des concepts source et cible. Le résultat correspond à l'état de configuration du concept cible modifié par une ou plusieurs actions des règles de restriction lorsque les états de configuration des règles correspondantes sont inclus dans l'état de configuration du concept source.

On définit $applyRF(\mathcal{RF}, \mathcal{S}_{source}, \mathcal{S}_{cible}) \rightarrow applyRule(SR_s, \mathcal{S}_{source}, \mathcal{S}_{cible})$ avec $\mathcal{RF} = \langle \mathcal{D}_{source}, \mathcal{D}_{cible}, SR \rangle$ et $SR_s = \{\bigcup \mathcal{R}_i | \mathcal{R}_i \in SR$ pour $\mathcal{R}_i = \langle \mathcal{S}_i, \mathcal{A}[fm]_i \rangle$ avec $\mathcal{S}_i \subseteq \mathcal{S}_{source}\}$.

Les fonctions de restriction sont contenues au sein des associations (voir Définition 5.2.4), une fonction de restriction ne peut donc travailler que sur les concepts correspondants aux extrémités de l'association.

Propriété 5.4.9 (Cohérence des fonctions de restriction au sein des associations) :
Soit $\mathcal{A}_{ij} = \{\mathcal{E}_i, \mathcal{E}_j, SRF\}$ une association où SRF représente l'ensemble des fonctions de restriction.

Si on a $\mathcal{A}_{ij} \rightsquigarrow (\mathcal{D}_i, \mathcal{D}_j)$ alors $\forall \mathcal{RF} \in SRF$, on a $\mathcal{RF} = \langle \mathcal{D}_i, \mathcal{D}_j, SR \rangle$ ou $\mathcal{RF} = \langle \mathcal{D}_j, \mathcal{D}_i, SR \rangle$.

Par ailleurs, les règles sont toutes interprétées lors de l'application d'une fonction, comme il n'existe pas de relation d'ordre dans l'application des actions sur un FM (voir Propriété 5.4.4), il n'existe donc pas non plus de relation d'ordre sur l'application des règles ni, par extension, sur l'application des fonctions de restriction.

Exemple 12 : Fonction de restriction

On considère dans notre exemple fil rouge une fonction de restriction entre *Appartement* et *Piece*.

Cette fonction définit trois règles pour respectivement *(i)* sélectionner le *Capteur* de *Temperature* lorsque la feature *Temperature* de l'*Ordinateur Central* est sélectionnée, *(ii)* sélectionner le *Capteur* de *Luminosité* lorsque la feature *Luminosité* de l'*Ordinateur Central* est sélectionnée et *(iii)* sélectionner le capteur de *Présence* lorsque la feature *Securite* de l'*Ordinateur Central* est sélectionnée.

Listing 5.9 – Fonction de restriction entre Appartement et Piece

```
regle appartement_piece_temperature
source Appartement
cible Piece
selections [Temperature], exclusions [] => SELECTIONNE CTemperature

regle appartement_piece_luminosite
source Appartement
cible Piece
selections [Luminosite], exclusions [] => SELECTIONNE CLuminosite

regle appartement_piece_securite
source Appartement
cible Piece
selections [Securite], exclusions [] => SELECTIONNE CPresence
```

5.4.3 Cohérence d'une fonction de restriction

Une fonction de restriction est définie par un ensemble de règles de restriction. On dit que la fonction est *cohérente* si elle ne contient pas des règles qui peuvent impliquer des actions contradictoires à partir d'états dont l'union est un état qui n'est pas faux.

Propriété 5.4.10 (Cohérence d'une fonction de restriction) :
Soit $\mathcal{RF}_{source \to cible} = \langle \mathcal{D}_{source}, \mathcal{D}_{cible}, SR \rangle$ une fonction de restriction.

On dit que $\mathcal{RF}$ est *cohérente* si en considérant $\forall\ SR_i = \{\mathcal{R}_0, ..., \mathcal{R}_n\} \subset SR$ telle que $SS_i = \{\bigcup_{k=0}^{n} \mathcal{S}_k | \mathcal{R}_k \in SR_i$ avec $\mathcal{R}_k = \langle \mathcal{S}_k, \mathcal{A}[fm]_k \rangle\}$ n'est pas *faux* (voir Propriété 5.3.4), alors $SA = \{\mathcal{A}[fm]_k | \mathcal{R}_k \in SR_i$ avec $\mathcal{R}_k = \langle \mathcal{S}_k, \mathcal{A}[fm]_k \rangle\}$ n'est pas *contradictoire* (voir Propriété 5.4.2).

5.4.4 Compatibilité des états de configuration

Nous définissons la notion de compatibilité des états de configuration : deux états de configuration sont *compatibles* s'il existe une association entre les concepts qu'ils représentent et s'ils sont compatibles selon les fonctions de restriction de cette association.

Nous commençons donc par définir la compatibilité de deux états de configuration par rapport à une fonction de restriction. Une fonction de restriction étant orientée, nous déterminons à partir de notre couple d'états de configuration, un état source de la fonction et un état cible qui ne sera pas utilisé dans la fonction. Nous définissons la compatibilité entre les deux états de configuration par le fait que l'application de la fonction retourne exactement l'état cible d'origine, en prenant pour arguments l'état source et l'état cible donnés.

Propriété 5.4.11 (Compatibilité de deux états de configuration par rapport à une fonction de restriction) :
Soit $\mathcal{RF}_{source \to cible} = \langle \mathcal{D}_{source}, \mathcal{D}_{cible}, SR \rangle$ une fonction de restriction.

Soit $\mathcal{S}_{source}$ un état de configuration du concept $\mathcal{D}_{source}$ et $\mathcal{S}_{cible}$ un état de configuration du concept $\mathcal{D}_{cible}$.

Le couple $(\mathcal{S}_{source}, \mathcal{S}_{cible})$ est *compatible* par rapport à $\mathcal{RF}_{source \to cible}$ si et seulement si : $applyRF(\mathcal{RF}_{source \to cible}, \mathcal{S}_{source}, \mathcal{S}_{cible}) \to \mathcal{S}_{cible}$.

On note alors : $(\mathcal{S}_{source}, \mathcal{S}_{cible}) \rightsquigarrow \mathcal{RF}_{source \to cible}$.

En conséquence de cette propriété, si $\mathcal{S}_i$ un état de configuration du concept $\mathcal{D}_{cible}$, alors $applyRF(\mathcal{RF}_{source \to cible}, \mathcal{S}_{source}, \mathcal{S}_i) \to \mathcal{S}'_i$ avec $(\mathcal{S}_{source}, \mathcal{S}'_i) \rightsquigarrow \mathcal{RF}_{source \to cible}$.

Exemple 13 : Compatibilité des états de configuration

Soit S_{appart} un état de configuration relatif au FM *Appartement* spécifiant que *Luminosité* est sélectionnée et S_{piece} un état de configuration relatif au FM *Piece* spécifiant que la feature *CLuminosité* est exclue.

D'après la fonction de restriction exprimée dans l'exemple 12 ces deux états de configuration sont incompatibles car ils ne respectent pas l'une des règles de restriction de la fonction.

La compatibilité d'un couple d'états de configuration par rapport à une association est déterminée par la compatibilité du couple par rapport à toutes les fonctions de restriction de l'association.

Propriété 5.4.12 (Compatibilité de deux états de configuration par rapport une association) :
Soit $\mathcal{A}_{ij}$ une association entre les concepts $\mathcal{D}_i$ et $\mathcal{D}_j$, $\mathcal{A}_{ij} = \langle \mathcal{E}_i, \mathcal{E}_j, SF \rangle$.

Soit $\mathcal{S}_i$ un état de configuration du concept $\mathcal{D}_i$ et $\mathcal{S}_j$ un état de configuration du concept $\mathcal{D}_j$.

Le couple $(\mathcal{S}_i, \mathcal{S}_j)$ est *compatible* par rapport à l'association $\mathcal{A}_{ij}$ si et seulement si :

$\forall\, \mathcal{RF}_{i\to j} \in SF$, $(\mathcal{S}_i, \mathcal{S}_j) \rightsquigarrow \mathcal{RF}_{i\to j}$ et $\forall\, \mathcal{RF}_{j\to i} \in SF$, $(\mathcal{S}_j, \mathcal{S}_i) \rightsquigarrow \mathcal{RF}_{j\to i}$.

On note la compatibilité d'un couple d'états de configuration par rapport à une association : $(\mathcal{S}_i, \mathcal{S}_j) \rightsquigarrow \mathcal{A}_{ij}$.

5.4.5 Règles contraposées et fonctions inverses

Si les associations sont bidirectionnelles, les règles et les fonctions de restriction en revanche sont orientées : elles référencent un concept source et un concept cible sur lesquels s'appliquer. Nous définissons donc la notion de *contraposée* à une règle et par extension de *fonction inverse* à une fonction, comme les règles et fonctions définissant une orientation contraire avec une sémantique équivalente.

L'idée est de pouvoir définir que si on a une règle du type "A implique B" alors on devrait également avoir le symétrique "non B implique non A". Dans notre formalisme cela se traduit par "état A implique action B" alors "non (état déduit de l'action B) implique non (action déduite de l'état A)".

Le calcul de la contraposée à une règle respecte les propriétés suivantes :

1. le FM source (resp. le FM cible) de la règle d'origine devient le FM cible (resp. le FM source) de la règle contraposée ;
2. l'action de la règle d'origine détermine l'état de configuration de la règle contraposée ;
3. l'état de configuration de la règle d'origine détermine l'action à effectuer dans la règle contraposée.

Propriété 5.4.13 (Calcul de la contraposée à une règle) :
Soit $\mathcal{R} = \langle \mathcal{S}, \mathcal{A}[fm] \rangle$ une règle de restriction avec $\mathcal{S} = \langle \mathcal{V}_s, SS, SE \rangle$ et $\mathcal{A}[fm] = \langle type, \mathcal{V}_c, element \rangle$.

Nous définissons $SR_{inv} = \{\mathcal{R}_{inv_0}, ..., \mathcal{R}_{inv_n}\}$ l'ensemble des règles contraposées, avec $\forall\, \mathcal{R}_{inv_i} \in SR_{inv}$, $\mathcal{R}_{inv_i} = \langle \mathcal{S}_{inv}, \mathcal{A}[fm]_{inv_i} \rangle$. Nous avons alors $\mathcal{S}_{inv} = \langle \mathcal{V}_c, SS_{inv}, SE_{inv} \rangle$ et $\mathcal{A}[fm]_{inv} = \langle type_{inv}, \mathcal{V}_s, element_{inv} \rangle$.

L'action d'origine $\mathcal{A}[fm]$ va permettre de déterminer l'état de configuration de la contraposée. Trois cas sont à envisager :

1. soit l'action est un ajout de contrainte, $type = AjouteContrainte$ l'état de configuration de la contraposée sera alors construit en considérant la contrainte : $element = \langle ST, SN \rangle$ où ST représente l'ensemble des littéraux sans opérateur et SN l'ensemble des features avec l'opérateur *non* ; on a alors pour l'état de configuration : $SS_{inv} = \{\bigcup \mathcal{F}_i | \mathcal{F}_i \in SN\}$ et $SE_{inv} = \{\bigcup \mathcal{F}_i | \mathcal{F}_i \in ST\}$;
2. soit l'action est une sélection $type = Selectionne$ l'état de configuration est alors $SS_{inv} = \emptyset$ et $SE_{inv} = \{element\}$;

3. soit l'action est une exclusion $type = Exclut$ l'état de configuration est alors $SS_{inv} = \{element\}$ et $SE_{inv} = \emptyset$.

Enfin, l'état de configuration d'origine $\mathcal{S}$ va déterminer l'action à effectuer. Nous devons considérer deux cas :

1. soit l'état de configuration ne concerne qu'une unique feature : $|SS \cup SE| = 1$, alors on crée l'action en fonction de l'ensemble où se trouve la feature : si $SS = \{\mathcal{F}_i\}$ (resp. SE) alors on a $\mathcal{A}[fm]_{inv} = \langle Exclut, \mathcal{V}_i, \mathcal{F}_i \rangle$ (resp. $\langle Selectionne, \mathcal{V}_i, \mathcal{F}_i \rangle$) ;
2. soit l'état de configuration concerne plusieurs features : $|SS \cup SE| > 1$, on crée alors une action de contrainte $\mathcal{A}[fm]_{inv} = \langle AjouteContrainte, \mathcal{V}_0, \mathcal{CST} \rangle$ avec $\mathcal{CST} = \langle ST, SN \rangle$ et $ST \equiv SE$ et $SN \equiv SS$.

Exemple 14 : Règles de restrictions et contraposées

Soit la règle entre *Appartement* et *Piece* telle que si la feature *Luminosité* est sélectionnée dans *Appartement*, elle l'est aussi dans *Piece* (voir 12).

La règle contraposée à cette règle est réalisée entre *Piece* et *Appartement* et exprime que si *Luminosité* est exclue dans *Piece* alors elle est aussi exclue dans *Appartement*.

Listing 5.10 – Règle de restriction contraposée entre Piece et Appartement

```
regle appartement_piece_luminosite
source Appartement
cible Piece
selections [Luminosite], exclusions [] => SELECTIONNE CLuminosite

regle appartement_piece_luminosite_contra
source Piece
cible Appartement
selections [], exclusions [CLuminosite] => EXCLU Luminosite
```

La règle contraposée pour la règle de restriction entre *Piece* et *Ouverture* pour la gestion des *Store* (voir exemple 11) donne lieu à une partie gauche bien plus complexe :

Listing 5.11 – Règle de restriction contraposée entre Ouverture et Piece

```
regle piece_ouverture_store
source Piece
cible Ouverture
selections [AStore], exclusions [] => AjouteContrainte Store or Porte

regle piece_ouverture_store_contra
source Ouverture
cible Piece
selections [], exclusions [Store, Porte] => EXCLU AStore
```

A l'inverse, la règle contraposée pour la règle de restriction entre *Piece* et *Appartement* pour le chauffage par le *Plancher* (voir exemple 11) donne lieu à une partie droite bien plus complexe :

Listing 5.12 – Règle de restriction contraposée entre Appartement et Piece

```
regle piece_appartement_plancher
source Piece
cible Appartement
selections [AThermostat], exclusions [Radiateur] => SELECTIONNE Plancher

regle piece_appartement_plancher_contra
source Appartement
cible Piece
```

```
selections [], exclusions [Plancher] => AjouteContrainte !AThermostat or Radiateur
```

Une fonction inverse à une fonction est donc construite par l'agrégation de toutes les règles contraposées des règles de restriction de la fonction d'origine.

Propriété 5.4.14 (Fonction inverse à une fonction de restriction) :
Soit $\mathcal{RF} = \langle \mathcal{D}_{source}, \mathcal{D}_{cible}, SR \rangle$ une fonction de restriction.

La fonction de restriction $\mathcal{RF}_{inv}$ est une fonction *inverse* de $\mathcal{RF}$ si et seulement si : $\mathcal{RF}_{inv} = \langle \mathcal{D}_{cible}, \mathcal{D}_{source}, SR_{inv} \rangle$ avec $SR_{inv} = \{\bigcup SR_i | \mathcal{R}_i \in SR$, et SR_i est l'ensemble des règles contraposées calculées à partir de $\mathcal{R}_i\}$.

5.5 Représentation d'une configuration composite

Une grande part de la flexibilité permise par notre processus de configuration réside dans la possibilité pour l'utilisateur de décider lui-même de la manière dont il veut composer les différentes configurations réalisées pour chacun des concepts représentés dans la ligne.

5.5.1 Définitions

Nous définissons la notion de **configuration composite** comme un assemblage de **sous-configurations** liées entre elles et respectant le modèle du domaine.

Une **sous-configuration** est à la fois une instance d'un concept du domaine mais également une configuration du FM représentant ce concept. Elle est ainsi définie par le concept qu'elle représente et par un état de configuration cohérent et complet.

Définition 5.5.1 (Sous-configuration) :
Nous définissons une sous-configuration $\mathcal{SC}$ comme un triplet $\langle \mathcal{D}, \mathcal{S}, \mathcal{C} \rangle$ avec $\mathcal{D}$ le concept qu'elle représente, $\mathcal{S}$ son état de configuration, et $\mathcal{C}$ le contexte de configuration dans lequel la sous-configuration été créée (nous définissons ce concept dans le chapitre suivant section 6.3).

En considérant $\mathcal{V}$ le FM référencé par $\mathcal{D}$, alors $\mathcal{S}$ est un état de configuration de $\mathcal{V}$. De plus, $\mathcal{S}$ représente un état de configuration complet (Propriété 5.3.2) et cohérent (Propriété 5.3.3) de $\mathcal{V}$.

On note la relation entre une sous-configuration et le concept qu'elle instancie : $\mathcal{SC} \rightsquigarrow \mathcal{D}$.

Le formalisme graphique que nous utilisons pour représenter une sous-configuration indique le nom du concept qui est instancié (le *type* dans la légende), le nom de l'instance réalisée afin de pouvoir l'identifier plus facilement et présente les features qui ont été sélectionnées dans la configuration. Toutes les autres features du FM qui ne sont pas représentées dans la sous-configuration sont de fait exclues.

Exemple 15 : Sous-configuration

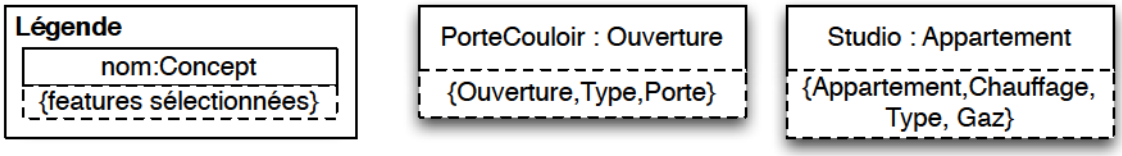


FIGURE 5.6 – Exemples de sous-configurations

La Figure 5.6 donne deux exemples très simples de sous-configurations de *Ouverture* et *Appartement* pour notre exemple fil rouge.

Les **liens** réalisés entre les sous-configurations sont des instances des associations entre les concepts représentés dans le modèle du domaine.

Définition 5.5.2 (Lien) :
Nous définissons un lien $\mathcal{L}$ comme un triplet $\langle \mathcal{SC}_i, \mathcal{SC}_j, \mathcal{A} \rangle$ où $\mathcal{SC}_i$ représente la sous-configuration d'une extrémité du lien, $\mathcal{SC}_j$ la sous-configuration de la seconde extrémité du lien, et $\mathcal{A}$ l'association instanciée par le lien.

Si $\mathcal{A} \rightsquigarrow \{\mathcal{D}_a, \mathcal{D}_b\}$, alors $\mathcal{SC}_i \rightsquigarrow \mathcal{D}_a$ et $\mathcal{SC}_j \rightsquigarrow \mathcal{D}_b$ ou inversement.

On note avec le symbole $\in$ la relation d'appartenance d'une sous-configuration à un lien. Par exemple : $\mathcal{SC}_i \in \mathcal{L}$.

Finalement la **configuration composite** est représentée comme un ensemble de sous-configurations et de liens entre ces sous-configurations, cet assemblage décrivant une instance du modèle du domaine.

Définition 5.5.3 (Configuration composite) :
Nous définissons une configuration composite $\mathcal{CC}$ comme un triplet $\langle \mathcal{M}, SSC, SL \rangle$ avec $\mathcal{M}$ le modèle du domaine que l'on cherche à satisfaire, $SSC = \{\mathcal{SC}_0, ..., \mathcal{SC}_n\}$ un ensemble de sous-configurations et $SL = \{\mathcal{L}_0, ..., \mathcal{L}_m\}$ un ensemble de liens entre ces sous-configurations.

On note la relation entre une configuration composite et le modèle du domaine qu'elle instancie : $\mathcal{CC} \rightsquigarrow \mathcal{M}$.

Exemple 16 : Configuration composite

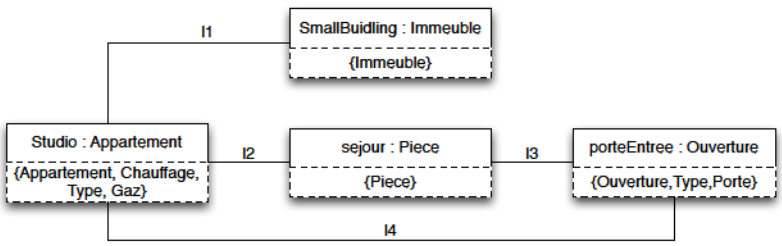


FIGURE 5.7 – Exemple d'une configuration composite minimale

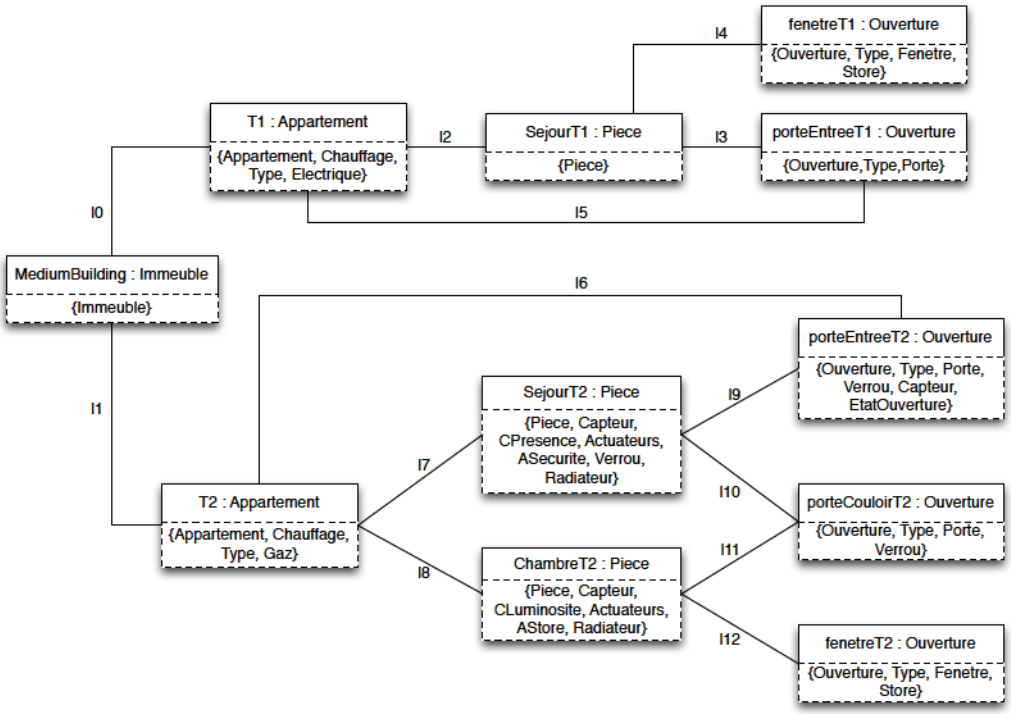


FIGURE 5.8 – Exemple d'une configuration composite

La Figure 5.7 présente une configuration composite extrêmement simple pour l'exemple fil rouge. Les liens sont représentés par un simple trait entre les sous-configurations, le label au dessus du lien ne fait pas partie du formalisme mais est donné afin de nous permettre de référencer le lien dans les explications. On la considère comme minimale au sens où les 4 concepts du modèle du domaine de notre exemple ainsi que les 4 associations sont instanciés une seule fois chacun.

En revanche, la Figure 5.8 présente toujours une configuration composite pour notre exemple fil rouge, mais cette fois avec de multiples instanciations pour les concepts d'*Appartement*, *Piece* et *Ouverture*, ainsi que pour toutes les associations du modèle du domaine.

5.5.2 Validité de la configuration composite

Plusieurs propriétés doivent être satisfaites pour s'assurer de la cohérence d'une configuration composite par rapport à son modèle du domaine.

On s'intéresse en premier lieu au respect de la multiplicité exprimée sur les concepts. Cette multiplicité est vérifiée en regardant la cardinalité des ensembles de sous-configurations qui instancient les différents concepts.

Propriété 5.5.1 (Validité de la multiplicité des concepts) :
Soit $\mathcal{M}$ un modèle du domaine. On a $\mathcal{M} = \langle SC, SA \rangle$.

Soit $\mathcal{CC} = \langle \mathcal{M}, SSC, SL \rangle$ une configuration composite.

On dit que la multiplicité des concepts est respectée dans $\mathcal{CC}$ si :
$\forall \mathcal{D}_i \in SC$, soit $SSC_i = \{\mathcal{SC}_i | \mathcal{SC}_i \in SSC, \mathcal{SC}_i \rightsquigarrow \mathcal{D}_i\}$, en considérant $\mathcal{D}_i = \langle id, \mathcal{V}_i, \mathcal{U}_i \rangle$ on a $|SSC_i| \in \mathcal{U}_i$.

Exemple 17 : Validité de la multiplicité des concepts

Ainsi la multiplicité des concepts est respectée dans les deux configurations composites présentées dans l'exemple 16 par rapport à leur modèle du domaine que l'on retrouve Figure 5.3.

En effet, le modèle du domaine impose un minimum d'une instance de chaque concept et il impose également que le concept d'*Immeuble* ne soit instancié qu'une unique fois.

Dans le cas de la configuration composite minimale ces restrictions sont bien respectées : chaque concept étant instancié une et une seule fois. Dans le cas de la seconde configuration composite, le concept d'*Appartement* est instancié deux fois, celui de *Piece* trois fois, et celui d'*Ouverture* cinq fois. Cependant, l'*Immeuble* reste instancié une seule fois, la configuration composite est donc également valide du point de vue des multiplicités sur les concepts.

On s'intéresse ensuite à la validation des multiplicités exprimées sur les associations du modèle du domaine, au sein de la configuration composite. Ainsi, on vérifie le respect de la multiplicité des associations, par la cardinalité des liens de la configuration composite.

Plus précisément, une association possède une multiplicité par extrémité : on doit donc vérifier la cardinalité pour chaque extrémité. Cela revient à être capable de calculer des ensembles regroupant toutes les sous-configurations qui correspondent au concept de l'extrémité que l'on doit vérifier et qui sont impliquées dans un lien correspondant à l'association que l'on vérifie. Une fois cet ensemble obtenu, il suffit de vérifier la validité de sa cardinalité.

Propriété 5.5.2 (Validité de la multiplicité d'une association) :
Soit $\mathcal{M}$ un modèle du domaine, $\mathcal{M} = \langle SC, SA \rangle$ où SC est un ensemble de concepts et SA un ensemble d'associations.

Soit $\mathcal{CC} = \{\mathcal{M}, SSC, SL\}$ une configuration composite.

Soit $\mathcal{A}_{ij} \in SA$ une association. On a $\mathcal{A}_{ij} = \langle \mathcal{E}_i, \mathcal{E}_j, SF \rangle$ avec $\mathcal{E}_i = \langle \mathcal{D}_i, \mathcal{U}_i \rangle$ une extrémité où $\mathcal{D}_i \in SC$ représente le concept référencé et $\mathcal{U}_i$ la multiplicité portée par ce côté de l'association. De la même façon, on a : $\mathcal{E}_j = \langle \mathcal{D}_j, \mathcal{U}_j \rangle$.

Soit SLA un ensemble regroupant les liens qui instancient cette association : $SLA = \{\bigcup \mathcal{L}_k | \mathcal{L}_k \in SL$ tel que $\mathcal{L}_k \rightsquigarrow \mathcal{A}_{ij}\}$.

Chaque lien lie deux sous-configurations, on peut créer un ensemble regroupant toutes ces sous-configurations : $SLC = \{\bigcup(\mathcal{SC}_u, \mathcal{SC}_v) | \mathcal{L}_k \in SLA$ tel que $\mathcal{L}_k = \langle \mathcal{SC}_u, \mathcal{SC}_v, \mathcal{A}_{ij} \rangle\}$.

Pour chacune des sous-configurations de SLC nous pouvons construire un sous-ensemble de SLC contenant les sous-configurations liées par un lien contenu dans SLA : $\forall \mathcal{SC}_i \in SLC$, $\exists SLC_i \subset SLC \setminus \mathcal{SC}_i$ tel que $SLC_i = \{\bigcup \mathcal{SC}_k | \mathcal{SC}_k \in SLC \setminus \mathcal{SC}_i$, tel que $(\mathcal{SC}_i, \mathcal{SC}_k) \in \mathcal{L}_k \in SLA\}$.

Comme SLA ne contient que les liens d'une association donnée, le sous-ensemble construit ne contient nécessairement que des sous-configurations représentant un même concept, qui ne peut pas être le même que celui de la sous-configuration utilisée pour construire le sous-ensemble (pour rappel, nous n'autorisons pas les associations réflexives dans le modèle du domaine).

Ainsi, vérifier la validité de la cardinalité de l'association pour la sous-configuration, revient à vérifier la cardinalité du sous-ensemble calculé, par rapport à la multiplicité portée par l'extrémité de l'association correspondant au concept du sous-ensemble.

Ainsi, $\forall \mathcal{SC}_i \in SLC$, SLC_i étant le sous-ensemble calculé à partir de $\mathcal{SC}_i$, et en considérant $\mathcal{SC}_i \rightsquigarrow \mathcal{D}_i$ et $\forall \mathcal{SC}_k \in SLC_i$, $\mathcal{SC}_k \rightsquigarrow \mathcal{D}_j$, alors la cardinalité de $\mathcal{SC}_i$ suivant l'association $\mathcal{A}_{ij}$

est respectée si et seulement si : $|SLC_i| \in \mathcal{U}_j$.

Pour que la multiplicité de l'association soit respectée il faut que la propriété soit vérifiée pour toutes les sous-configurations liées par une instance de l'association. On peut, par ailleurs, utiliser cette propriété en ne considérant que la borne inférieure ou supérieure des multiplicités, on dit alors que la propriété est respectée pour une borne inférieure ou supérieure.

Exemple 18 : Validité de la multiplicité des liens

La configuration composite représentée dans la Figure 5.8 est valide du point de vue de la cardinalité des liens par rapport à son modèle du domaine représenté dans la Figure 5.3.

En effet, les 4 associations sont instanciées de la façon suivante :

— entre *Immeuble* et *Appartement* : {l0, l1} ;
— entre *Appartement* et *Ouverture* : {l5, l6} ;
— entre *Appartement* et *Piece* : {l2, l7, l8} ;
— entre *Piece* et *Ouverture* : {l3, l4, l9, l10, l11, l12}.

Si l'on se concentre sur la dernière association, on constate qu'elle permet de relier 8 des 11 sous-configurations. On peut donc construire l'ensemble des sous-configurations impliquées dans cette association : {SejourT1, fenetreT1, porteEntreeT1, SejourT2, porteEntreeT2, porteCouloirT2, ChambreT2, fenetreT2}.

Nous devons maintenant tester les cardinalités pour chacune des 8 sous-configurations.

SejourT1 Cette sous-configuration est liée aux sous-configurations {fenetreT1, porteEntreeT1}. La cardinalité de cet ensemble est de 2 éléments. Comme *SejourT1* est une instance de *Piece*, nous devons comparer la cardinalité de l'ensemble à la multiplicité portée par l'association sur l'extrémité *Ouverture* : on constate que cette multiplicité est `1..*`. La sous-configuration *SejourT1* respecte donc bien la cardinalité de l'association entre *Piece* et *Ouverture*.

fenetreT1 Cette sous-configuration n'est liée qu'à la sous-configuration *SejourT1*. La cardinalité est donc de 1 élément. Cette fois la sous-configuration est une instance de *Ouverture* nous devons donc comparer la cardinalité de l'ensemble à la multiplicité portée par l'association sur l'extrémité *Piece* : cette multiplicité est également `1..*`. La sous-configuration *fenetreT1* respecte donc également bien la cardinalité de l'association entre *Piece* et *Ouverture*.

On peut calculer ainsi toutes les cardinalités et constater qu'elles respectent toutes les multiplicités prescrites.

On considère la validité des configurations d'un lien par leur compatibilité définie par les fonctions de restriction portées par l'association qu'instancie le lien.

Propriété 5.5.3 (Validité des configurations d'un lien) :
Soit $\mathcal{L} = \langle \mathcal{SC}_a, \mathcal{SC}_b, \mathcal{A}_{ab} \rangle$ un lien avec $\mathcal{SC}_a = \langle \mathcal{D}_a, \mathcal{S}_a, \mathcal{C}_a \rangle$ et $\mathcal{SC}_b = \langle \mathcal{D}_b, \mathcal{S}_b, \mathcal{C}_b \rangle$ les deux sous-configurations qu'il relie et $\mathcal{A}_{ab}$ l'association qu'il instancie.

Soit $\mathcal{A}_{ab} = \langle id, \mathcal{E}_a, \mathcal{E}_b, SF \rangle \in SA$ où $\mathcal{E}_a$ et $\mathcal{E}_b$ représentent les extrémités de l'association

sur $\mathcal{D}_a$ et $\mathcal{D}_b$ et SF est l'ensemble des fonctions de restriction. Comme nous l'avons défini dans la sous-section 5.4.5 l'association contient pour chaque fonction sa fonction inverse. Les propriétés sur les fonctions inverses nous permettent de ne vérifier la validité du lien que dans un seul sens.

Soit $\mathcal{E}_a = \langle \mathcal{D}_a, \mathcal{U}_a \rangle$ et $\mathcal{E}_b = \langle \mathcal{D}_b, \mathcal{U}_b \rangle$.

Soit l'ensemble des fonctions de restriction $SF_{a\to b} = \{\mathcal{RF}_0, ..., \mathcal{RF}_n\} \subset SF$ tel que $\forall\, \mathcal{RF}_i \in SF_{a\to b}$, $\mathcal{RF}_i = \langle \mathcal{D}_a, \mathcal{D}_b, SR \rangle$ avec $\mathcal{D}_a$ le concept source et $\mathcal{D}_b$ le concept cible.

$\mathcal{S}_a$ et $\mathcal{S}_b$ étant les états de configurations respectifs de $\mathcal{SC}_a$ et $\mathcal{SC}_b$, ces deux sous-configurations sont valides relativement au lien $\mathcal{L}$ si et seulement si
$\forall\, \mathcal{RF}_i \in SF_{a\to b}$, $applyRF(\mathcal{RF}_i, \mathcal{S}_a, \mathcal{S}_b) \to \mathcal{S}_b$.

Une configuration composite est dite *cohérente* si les multiplicités maximum du modèle du domaine sont respectées, aussi bien au niveau *(i)* des concepts que *(ii)* des associations, et si *(iii)* les sous-configurations liées sont compatibles.

Propriété 5.5.4 (Configuration composite cohérente) :
Soit $\mathcal{M}$ un modèle du domaine, $\mathcal{M} = \langle SC, SA \rangle$ où SC est un ensemble $\mathcal{D}_0, ..., \mathcal{D}_n$ de concepts et SA un ensemble $\mathcal{A}_0, ..., \mathcal{A}_m$ d'associations.

Soit $\mathcal{CC} = \langle \mathcal{M}, SSC, SL \rangle$ une configuration composite.

$\mathcal{CC}$ est dite cohérente si :

1. $\forall\, \mathcal{D} = \langle id, \mathcal{V}, \mathcal{U} \rangle \in SC$, avec $\mathcal{U} = \langle \mathcal{L}^-, \mathcal{L}^+ \rangle$ la multiplicité de $\mathcal{D}$, soit $S = \{\mathcal{SC}_0, ..., \mathcal{SC}_n\}$ tel que $S \subseteq SSC$ et $\mathcal{SC}_i \rightsquigarrow \mathcal{D}$ pour $0 \leqslant i \leqslant n$ alors $|S| \leqslant \mathcal{L}^+$;
2. la propriété de validité de la cardinalité des liens de la configuration composite est réalisée sur la borne supérieure (voir Propriété 5.5.2) et
3. la propriété de validité des configurations des liens de la configuration composite est réalisée (voir Propriété 5.5.3).

Une configuration composite est dite *cohérente et terminée* si : *(i)* le graphe réalisé en considérant les sous-configurations comme des noeuds et les liens comme des arcs est connexe, si *(ii)* les multiplicités minimum et maximum du modèle du domaine sont respectées aussi bien au niveau des concepts que *(iii)* des associations, et si *(iv)* les sous-configurations liées sont compatibles.

Propriété 5.5.5 (Configuration composite cohérente et terminée) :
Soit $\mathcal{M}$ un modèle du domaine, $\mathcal{M} = \langle SC, SA \rangle$ où SC est un ensemble $\mathcal{D}_0, ..., \mathcal{D}_n$ de concepts et SA un ensemble $\mathcal{A}_0, ..., \mathcal{A}_m$ d'associations.

Soit $\mathcal{CC} = \langle \mathcal{M}, SSC, SL \rangle$ une configuration composite.

$\mathcal{CC}$ est dite cohérente et terminée si :

1. $\forall\, \mathcal{SC} \in SSC$, $\exists\, \mathcal{L} \in SL$ tel que $\mathcal{SC} \in \mathcal{L}$;
2. $\forall\, \mathcal{D} = \langle id, \mathcal{V}, \mathcal{U}, SA \rangle \in SC$, avec $\mathcal{U}$ la multiplicité de $\mathcal{D}$, soit $S = \{\mathcal{SC}_0, ..., \mathcal{SC}_n\}$ tel que $S \subseteq SSC$ et $\mathcal{SC}_i \rightsquigarrow \mathcal{D}$ pour $0 \leqslant i \leqslant n$ alors $|S| \in \mathcal{U}$;
3. la propriété de validité de la cardinalité des liens de la configuration composite est réalisée (voir Propriété 5.5.2) et
4. la propriété de validité des configurations des liens de la configuration composite est réalisée (voir Propriété 5.5.3).

Nous définissons par ailleurs la notion de *configuration composite minimale valide*.
Une configuration composite est dite *minimale* si :

1. chaque concept du domaine est instancié une et une seule fois ;
2. chaque association du domaine est instanciée une et une seule fois.

Propriété 5.5.6 (Configuration composite minimale valide) :
Soit $\mathcal{M}$ un modèle du domaine, $\mathcal{M} = \langle SC, SA \rangle$ où SC est un ensemble $\mathcal{D}_0, ..., \mathcal{D}_n$ de concepts et SA un ensemble $\mathcal{A}_0, ..., \mathcal{A}_m$ d'associations.
Soit $\mathcal{CC} = \langle \mathcal{M}, SSC, SL \rangle$ une configuration composite.
$\mathcal{CC}$ est dite minimale et valide si :

1. $\forall\, \mathcal{D}_i \in SC, \exists\, !\mathcal{SC}_i \in SSC$ telle que $\mathcal{SC}_i \rightsquigarrow \mathcal{D}_i$;
2. $\forall\, \mathcal{A}_i \in SA, \exists\, !\mathcal{L}_i \in SL$ tel que $\mathcal{L}_i \rightsquigarrow \mathcal{A}_i$;
3. $\mathcal{CC}$ est cohérente et terminée selon la Propriété 5.5.5.

5.6 Cohérence de l'ensemble des informations du domaine

Nous avons défini dans les sections précédentes les éléments du formalisme permettant à la fois de définir un modèle du domaine représentant les différents concepts de la ligne de produit logiciel ainsi que leur variabilité ; mais permettant aussi de définir une configuration composite qui soit conforme avec le modèle du domaine.

Cependant, pour qu'une telle configuration composite puisse être réalisée, il est nécessaire que l'ensemble des informations du domaine soient cohérentes entre elles. Nous définissons ainsi la cohérence d'une LPL selon plusieurs critères :

1. le modèle du domaine de la LPL doit être cohérent, c'est à dire connexe (Propriété 5.2.5), cohérent au niveau des concepts et des associations (Propriété 5.2.6) et cohérent au niveau des multiplicités (Propriété 5.2.7) ;
2. tous les modèles de variabilités référencés dans le modèle du domaine doivent être cohérents (Propriété 5.3.1) ;
3. toutes les fonctions de restriction portées par les associations du modèle du domaine doivent être cohérentes (Propriété 5.4.10) ;
4. chaque configuration complète (Propriété 5.3.2) et cohérente (Propriété 5.3.3) de chaque FM du modèle du domaine doit pouvoir être incluse dans une configuration composite minimale complète et valide (Propriété 5.5.6).

Le dernier critère est un critère de réalisabilité : il détermine que toutes les configurations de tous les FM employés dans le modèle du domaine pourront être utilisées dans au moins une configuration composite complète et valide.

Ce critère nous permet de garantir par la suite que notre algorithme de propagation (voir section 6.4) dirigera toujours l'utilisateur final vers une solution valide : il ne sera donc jamais bloqué dans un état ne lui permettant plus d'effectuer d'actions, à moins d'avoir atteint une configuration composite complète.

Propriété 5.6.1 (Réalisabilité d'une LPL) :
Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine.

Soit $SFM = \{\mathcal{V} | \mathcal{D} \in SC, \mathcal{D} = \langle id, \mathcal{V}, \mathcal{U} \rangle\}$ l'ensemble des FM inclus dans le modèle du domaine.

Soit $SC = \{\bigcup SC_i | \mathcal{V} \in SFM, SC_i = [\![\mathcal{V}]\!]\}$ l'ensemble des états de configuration valides de l'ensemble des FM du modèle du domaine.

On dit que la LPL modélisée par $\mathcal{M}$ est *réalisable* si et seulement si $\forall\, \mathcal{S} \in SC$, $\exists\, \mathcal{CC} = \langle \mathcal{M}, SSC, SL \rangle$ tel que $\mathcal{SC} = \langle \mathcal{D}, \mathcal{S}, \mathcal{C} \rangle \in SSC$ et $\mathcal{CC}$ est valide et complète (voir Propriété 5.5.5).

Par extension, nous parlons de la réalisabilité d'une sous-configuration pour signifier qu'elle peut être incluse dans une configuration composite minimale valide de la LPL.

Ainsi, la vérification de la cohérence de la LPL passe par la vérification de chacune des propriétés de manière indépendante. Nous montrons dans la suite comment cette vérification peut être réalisée pour chaque propriété et quel est le coût théorique de cette vérification.

5.6.1 Cohérence du modèle du domaine

La cohérence du modèle du domaine consiste à vérifier sa connexité, sa cohérence au niveau des concepts et des associations, ainsi que sa cohérence au niveau des multiplicités.

La connexité du modèle du domaine (Propriété 5.2.5) se vérifie simplement par un parcours de graphe [Lacomme 03]. Un algorithme de parcours en profondeur permet de vérifier la propriété avec une complexité en O(m) où m représente le nombre d'associations du modèle du domaine.

La cohérence des concepts et des associations (Propriété 5.2.6) peut se vérifier en itérant sur chaque concept et association du modèle du domaine et en s'assurant qu'ils référencent bien des éléments du même modèle du domaine : la complexité est donc en O($m + n$) où m représente le nombre d'associations et n le nombre de concepts.

La vérification de la cohérence des multiplicités nécessite également de parcourir l'ensemble des concepts et de vérifier les règles édictées dans la Propriété 5.2.7, sa complexité est donc la même que la précédente.

Les deux dernières propriétés nécessitant d'itérer sur les concepts, on peut imaginer un algorithme permettant d'effectuer l'ensemble des vérifications de manière optimale avec une complexité algorithmique linéaire sur le nombre de concepts, soit en O(n).

5.6.2 Cohérence des FM

L'analyse des FM a fait l'objet de nombreux travaux notamment pour détecter les incohérences des FM [Benavides 10]. Dans le cadre de notre formalisme (voir Propriété 5.3.1), nous souhaitons au minimum vérifier que le FM possède au moins une configuration et que chaque feature peut être impliquée dans au moins une configuration valide du FM (il n'existe alors aucune *feature morte*).

Cependant, vérifier que le FM contient au moins une configuration valide se ramène à un problème de satisfiabilité. On peut en effet représenter le FM sous la forme d'une formule CNF : vérifier l'existence d'au moins une configuration revient alors à trouver une solution à cette formule.

Or le problème de satisfiabilité reste un problème difficile d'algorithmique de type NP-complet : on ne peut théoriquement pas garantir que le problème pourra être résolu en un temps polynomial quelles que soient ses conditions initiales.

Malgré tout, de nombreux travaux ont permis de réaliser des algorithmes de résolution des problèmes de satisfiabilité utilisant des heuristiques très performantes. Ainsi Mendonca *et al.* ont montré que des solveurs SAT parviennent très bien à résoudre des formules CNF calculées à partir de FM possédant jusqu'à 10 000 features et possédant une forte densité (un grand nombre de contraintes par feature) avec des résultats de l'ordre de la seconde [Mendonca 09].

Ainsi, en utilisant un solveur SAT, il est en pratique possible de savoir en un temps raisonnable s'il existe une configuration valide pour un FM donné. En outre, calculer les features mortes d'un FM revient à vérifier pour chaque feature qu'il existe une solution à la formule CNF correspondante au FM si la feature est sélectionnée, soit si sa valeur vaut *vraie*. Cela revient donc à réaliser une vérification de satisfiabilité par feature. L'opération reste donc complexe en théorie, mais elle se résout en pratique en un temps proportionnel au nombre de features et au temps pris par une opération de calcul de satisfiabilité sur le FM.

5.6.3 Cohérence des fonctions de restriction

La vérification de la cohérence des fonctions de restriction selon la Propriété 5.4.10 nécessite de considérer chacune des fonctions indépendamment et de calculer l'ensemble des combinaisons de règles de restrictions possibles telles que les états de configurations soient cohérents. Dans le pire des cas, cela revient à calculer l'ensemble des combinaisons de règles de restriction possible soit pour R règles de restriction, $2^R - 1$ possibilités. La vérification de cette propriété sera donc effectuée avec une complexité majorée par 2^n où n représente l'ensemble des règles de restriction du modèle du domaine, soit une complexité en O(2^n).

5.6.4 Réalisabilité de la LPL

De manière naïve la vérification de la réalisabilité de la LPL définie dans la Propriété 5.6.1 nécessite de considérer indépendamment chacune des configurations possibles de chaque FM de la LPL et de chercher une configuration composite minimale valide l'incluant.

Cette vision naïve implique cependant une très grande complexité algorithmique. En effet, considérer indépendamment chacune des configurations possibles de chaque FM de la LPL implique un premier parcours de l'ensemble total de chacune des sous-configurations pour chacun des concepts. Ensuite, réaliser une configuration composite minimale incluant la sous-configuration considérée implique de trouver pour chacun des concepts une sous-configuration valide avec la première sous-configuration, mais également avec toutes les autres choisies.

En considérant que l'on connaît pour chaque concept l'ensemble des sous-configurations valides (ce qui n'est pas une opération triviale pour les solveurs comme discuté au-dessus), il est possible de déterminer la complexité de l'algorithme naïf en fonction du nombre de comparaisons à faire entre les sous-configurations. Cette complexité va s'exprimer en fonction du nombre de sous-configurations des différents concepts et en fonction du nombre d'associations du modèle du domaine.

En effet, si l'on considère un modèle du domaine $\mathcal{M}$ contenant l'ensemble des concepts $SC = \{\mathcal{D}_0, ..., \mathcal{D}_k\}$ et l'ensemble des associations $SA = \{\mathcal{A}_0, ..., \mathcal{A}_l\}$. L'algorithme naïf de réalisabilité considère une sous-configuration, par exemple relative au concept $\mathcal{D}_0$ et cherche des sous-configurations valides pour tous les concepts restants. Dans le pire des cas, cela revient pour cette sous-configuration de $\mathcal{D}_1$ à effectuer : $\prod_{i=1}^{i=k} |[\![\mathcal{V}_i]\!]|$ avec $\mathcal{V}_i \rightsquigarrow \mathcal{D}_i$. Cependant, ces comparaisons vont uniquement permettre de trouver les sous-configurations compatibles avec la sous-configuration de $\mathcal{D}_0$: il est nécessaire également de vérifier que les sous-configurations obtenues sont compatibles *entre-elles*.

En considérant $m = |SA|$, le nombre d'associations du modèle du domaine, la complexité en nombre de comparaisons de l'algorithme naïf de réalisabilité est alors dans le pire des cas : $m * \prod_{i=0}^{i=k} |[\![\mathcal{V}_i]\!]|$ avec $\mathcal{V}_i \rightsquigarrow \mathcal{D}_i$. Par ailleurs, les associations constituent les arêtes d'un graphe défini par le modèle du domaine dans lequel les concepts représentent les nœuds et les associations les arcs. Dans le pire des cas, le graphe est fortement connexe et son nombre d'arêtes est donc de : $\frac{k*(k-1)}{2}$ [Lacomme 03]. En outre, nous savons que quel que soit $SA = \{A_0, ..., A_x\}$ avec A_i un ensemble quelconque d'éléments, $\prod_{i=0}^{i=x} |A_i| \leqslant \frac{n^x}{x}$ avec $n = \sum_{i=0}^{i=x} |A_i|$.

L'algorithme naïf de test de la réalisabilité a donc une complexité en O($\frac{n^k}{k^{k-2}}$) où $n = \sum_{i=0}^{i=k} |[\![\mathcal{V}_i]\!]|$ représente le nombre de sous-configurations du système. Cette complexité est très grande particulièrement si le nombre de sous-configurations à considérer est grand. Nous proposons dans la section 6.8 un algorithme permettant de vérifier la réalisabilité d'une LPL de manière itérative, en se basant sur des propriétés du graphe défini par le modèle du domaine et en exploitant des outils définis pour le processus de configuration.

5.7 Conclusion

Nous avons présenté dans ce chapitre les éléments du formalisme nécessaires à la représentation de la variabilité dans une LPL complexe en conservant une forte séparation des préoccupations.

Cette représentation nous permet de conserver un modèle du domaine possédant un haut niveau d'abstraction, en considérant la LPL comme un ensemble de concepts et d'associations entre ces concepts, tout en réutilisant des modèles de variabilité pour chacun des concepts déterminé dans la LPL. Nous avons défini par le biais d'états et d'actions de configuration un modèle de contraintes entre ces modèles de variabilité, capable de calculer automatiquement les contraintes inverses afin de permettre des associations bidirectionnelles.

Nous avons défini également la formalisation d'une configuration composite réalisable pour une LPL complexe, et qui est constituée d'un assemblage de sous-configurations liées ensemble en respectant le formalisme décrit par le modèle du domaine.

Enfin, nous avons déterminé quelles sont les propriétés de cohérence à avoir dans cette modélisation de la LPL afin de garantir l'obtention d'une configuration composite terminée qui soit cohérente.

CHAPITRE 6

UN PROCESSUS FLEXIBLE DE CONFIGURATION

Sommaire

Résumé Nous défendons dans notre approche un processus de configuration qui soit à la fois flexible et cohérent. Cohérent, car dans une LPL nécessitant de faire de très nombreux choix, il est important que l'utilisateur soit assuré en permanence de ne pas faire d'erreurs. Flexible, car il nous semble important de laisser à l'utilisateur le maximum de marge de manœuvre afin de lui permettre d'atteindre un produit dans l'ordre dans lequel il l'entend. Nous définissons ainsi dans ce chapitre le formalisme et les outils nécessaires afin de garantir ces deux propriétés.

Nous expliquons en premier lieu en quoi il est indispensable dans notre approche d'avoir un environnement de raisonnement dynamique sur les FM. Nous formalisons ensuite ce que nous nommons un "contexte de configuration", qui est garant de la flexibilité, puis nous présentons un algorithme de propagation qui exploite les contextes afin de garantir la cohérence des choix utilisateurs. Puis nous définissons un processus de configuration par étapes, dans lequel les actions de l'utilisateur sont enregistrées afin qu'il puisse annuler n'importe quelle action précédente sans rompre la cohérence. Enfin, nous détaillons les différentes actions utilisateur et leur enchaînement afin de montrer comment elles permettent de respecter nos propriétés de cohérence et flexibilité. Nous illustrons ensuite l'ensemble du processus de configuration à travers notre exemple fil rouge. Pour terminer, nous présentons un algorithme de vérification de la réalisabilité qui est une hypothèse nécessaire à la bonne réalisation du processus de configuration, et qui utilise les outils définis pour ce même processus.

Nous tentons ainsi de répondre dans ce chapitre au second objectif de cette thèse, à savoir garantir la flexibilité et la cohérence du processus de configuration d'une LPL complexe (voir section 4.3) ainsi qu'à une partie du troisième qui concernait la garantie de l'utilisabilité du processus de configuration (voir section 4.4).

6.1 Introduction

Nous avons défini dans le chapitre précédent un formalisme permettant de représenter la variabilité et les configurations d'une ligne de produits logiciels complexes en conservant une séparation des préoccupations entre les informations liées à la représentation abstraite du domaine (les concepts, associations et multiplicités) et les informations relatives à la variabilité des features du domaine (les modèles de variabilités).

Cependant, cette représentation est inutilisable sans un processus de configuration adapté. Nous défendons dans ce chapitre une approche à la fois flexible et cohérente du processus de configuration.

La cohérence est nécessaire pour l'obtention d'une configuration composite terminée qui soit cohérente : un processus de configuration complexe ne devrait jamais permettre à l'utilisateur d'enfreindre les contraintes imposées par le modèle du domaine.

Par ailleurs, nous définissons la flexibilité d'un processus de configuration comme la capacité à atteindre n'importe quelle configuration composite cohérente à partir d'une configuration composite cohérente qu'elle inclut. Cela signifie que l'utilisateur doit être en mesure d'atteindre n'importe quelle configuration composite cohérente.

Propriété 6.1.1 (Cohérence et flexibilité du processus de configuration) :
Soit $\mathcal{M}$ un modèle du domaine. Soit $\mathbb{C}$ l'ensemble des configurations composites cohérentes réalisables à partir de $\mathcal{M}$ (voir Propriété 5.5.4).

Le processus de configuration est *cohérent* :

- si toutes les configurations composites construites à travers ce processus appartiennent à $\mathbb{C}$,
- s'il est toujours possible d'atteindre une configuration composite terminée (voir Propriété 5.5.5)
- et si toutes les configurations composites de $\mathbb{C}$ sont atteignables.

Le processus de configuration est *flexible* si $\forall \mathcal{CC}_i, \mathcal{CC}_j \in \mathbb{C}$ tels que $\mathcal{CC}_i \subseteq \mathcal{CC}_j$, alors il est possible de construire $\mathcal{CC}_j$ à partir de $\mathcal{CC}_i$.

L'objectif de ce chapitre est donc de présenter les éléments du formalisme et les algorithmes qui permettent de nous assurer que ces deux propriétés sont respectées. Pour cela, nous définissons dans un premier temps le raisonnement dynamique sur les FM qui est une base nécessaire à un processus de configuration dynamique (section 6.2). Nous présentons ensuite les éléments supplémentaires de notre formalisme relatifs à des contextes de configurations, permettant d'assurer la flexibilité du processus de configuration (section 6.3). Nous montrons par notre algorithme de propagation comment la cohérence des contextes est assurée (section 6.4). Par ailleurs, le processus de configuration est le fait de décisions réalisées par l'utilisateur afin de définir une configuration composite : nous définissons les différentes étapes du processus de configuration et sa formalisation à travers des actions utilisateur (section 6.5). Afin de garantir la cohérence et la flexibilité du processus de configuration, nous définissons les combinaisons d'actions utilisateur qui sont autorisées, ainsi que les mécanismes de vérification et d'application de certaines actions utilisateur (section 6.6). Nous illustrons dans l'avant-dernière section l'ensemble de notre contribution sur les processus de configuration en réutilisant notre exemple fil rouge (section 6.7).

Enfin, l'ensemble de notre contribution repose sur la définition d'une ligne de produits logiciels cohérente qui a été donnée dans le chapitre précédent (voir sous-section 5.6.4) :

nous proposons un algorithme permettant de vérifier la réalisabilité de la LPL qui exploite les différents outils formalisés dans ce chapitre et qui offre une complexité moindre que l'algorithme naïf (section 6.8).

6.2 Raisonnement dynamique sur les FM

Les FM sont utilisés pour modéliser la variabilité c'est-à-dire donner une représentation abstraite des éléments communs et variants d'un ensemble de logiciels appartenant à une même famille. Cette représentation est utile pour un architecte afin de définir l'ensemble des éléments, concepts ou composants disponibles au sein de la famille de logiciels. Mais elle est également utilisée par l'utilisateur final afin de configurer un nouveau produit.

La configuration d'un produit au sein d'un FM consiste donc, pour l'utilisateur, à sélectionner ou exclure des features jusqu'à l'obtention d'une configuration valide selon la sémantique du FM. Cependant, en regard du grand nombre de features possibles dans un FM, et donc du nombre encore plus grand de combinaisons possibles dans la sélection il est nécessaire d'utiliser des mécanismes d'inférences afin d'aider l'utilisateur dans ses choix.

Ainsi il est possible d'effectuer des inférences automatiques en calculant à partir d'un FM une formule booléenne dans laquelle chaque littéral est une feature et d'utiliser les choix des utilisateurs en considérant qu'une sélection correspond à une valuation d'un littéral comme *vrai* et une exclusion à une valuation comme *faux* [Mendonca 09]. Des algorithmes de satisfiabilité (SAT) sont alors utilisés pour déterminer s'il existe un modèle de solution *vraie* à la formule booléenne. En réalisant ce calcul, ces algorithmes mettent en évidence : *(i)* les littéraux pour lesquels il existe encore des choix possibles et *(ii)* les littéraux qui ne peuvent être valués que d'une seule façon afin que la formule soit vraie.

Exemple 19 : Raisonnement dynamique sur les FM

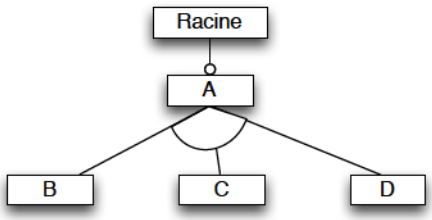


FIGURE 6.1 – Exemple d'un feature model très simple

Par exemple, dans le cas du FM représenté dans la Figure 6.1, il existe 4 configurations possibles :

- {Racine}
- {Racine, A, B}
- {Racine, A, C}
- {Racine, A, D}

La formule booléenne représentant ce FM est alors :
$\mathbf{F} = Racine \vee (Racine \wedge A \wedge (B \oplus C \oplus D))$ [a]

Dès lors un algorithme SAT peut directement inférer que $\mathbf{F} \models vrai \Rightarrow Racine = vrai$.

Ensuite, si la feature B est sélectionnée, alors la formule **F** est valuée en considérant B comme vraie : l'algorithme peut alors inférer que C et D doivent être exclus et que A doit être sélectionné pour que **F** soit vraie.

a. En logique booléenne, le symbole $\oplus$ représente la fonction XOR.

Dès lors n'importe quel choix de l'utilisateur peut mener à exclure ou sélectionner des features de manière complètement automatique en se conformant à la formule du FM. De nombreux travaux ont déjà été réalisés formalisant ces mécanismes de raisonnement et d'inférence, que nous considérons comme acquis dans le cadre de nos propres travaux [Mendonca 09, Schobbens 07].

Afin de permettre à l'utilisateur de construire une configuration composite cohérente et terminée (voir section 5.5) nous ajoutons à cette base les différentes notions qui nous seront utiles à travers le processus de configuration.

Nous définissons ainsi le concept d'**état du processus de configuration** (ConfigurationProcessState ou **CPS**). Un CPS représente une sous-configuration *en cours* de réalisation. Une fois la sous-configuration terminée, le CPS contiendra une référence vers celle-ci. Il est nécessaire dans le cadre de notre processus de configuration de conserver en permanence à la fois l'état de configuration d'un FM en train d'être configuré par l'utilisateur, mais également la liste des actions appliquées soit par l'utilisateur, soit de manière automatique par inférence du moteur de raisonnement.

Ainsi, nous avons 3 concepts distincts dans notre formalisme liés à la configuration d'un FM :

- un **état de configuration** est une représentation minimale d'une configuration partielle ou non d'un FM (voir Définition 5.3.4) ;
- une **sous-configuration** représente une configuration terminée d'un FM *mais aussi* une instance d'un concept du modèle du domaine (voir Définition 5.5.1) ;
- un **état du processus de configuration** (CPS) est une représentation du *processus de configuration* du FM et non pas de la configuration du FM elle-même : il est lié à un état de configuration, – puisqu'à chaque étape du processus on obtient un nouvel état de configuration –, possède un ensemble d'actions qui ont mené à cet état et référence la sous-configuration auquel il a mené s'il est terminé.

Un CPS est donc défini à la fois par le concept en train d'être configuré, l'état de configuration dans lequel il se trouve, la liste des actions effectuées sur le FM ainsi que la sous-configuration qu'il a permis de réaliser. Un CPS représente un état de configuration qui est en permanence cohérent selon la Propriété 5.3.3. La liste des actions effectuées du CPS est en cohérence avec son état de configuration, qui est lui même équivalent à l'état de configuration de la sous-configuration que le CPS référence.

Définition 6.2.1 (ConfigurationProcessState) :
Nous définissons un état du processus de configuration $\mathcal{CPS}$ comme un quadruplet $\langle \mathcal{D}, \mathcal{S}, SA, \mathcal{SC} \rangle$ où $\mathcal{D}$ représente le concept en train d'être configuré, $\mathcal{S}$ l'état de configuration, SA l'ensemble $\{\mathcal{A}[fm]_0, ..., \mathcal{A}[fm]_n\}$ des actions effectuées sur le FM et $\mathcal{SC}$ référence la configuration réalisée au sein de la configuration composite à partir de cet état du processus de configuration.

Un CPS est dit *terminé* si $\mathcal{SC} \neq \varnothing$: cela signifie alors qu'une configuration a été validée à partir de cette étape du processus de configuration et aucune action future ne pourra être réalisée au sein de ce CPS.

En considérant $\mathcal{V}$ le FM référencé par $\mathcal{D}$, et $\mathcal{CC}$ la configuration composite en train d'être réalisée, nous avons les propriétés suivantes :

— $\forall \mathcal{A}[fm]_i \in SA$, $\mathcal{A}[fm]_i$ s'applique sur $\mathcal{V}$,
— $\mathcal{S}$ est défini sur $\mathcal{V}$ et est cohérent selon la Propriété 5.3.3,
— $apply(SA, \langle \mathcal{V}, \emptyset, \emptyset \rangle) \rightarrow \mathcal{S}$,
— si $\mathcal{SC} \neq \varnothing$, $\mathcal{SC} \in \mathcal{CC}$ et $\mathcal{SC} = \langle \mathcal{D}, \mathcal{S}, \mathcal{C} \rangle$.

Nous pouvons définir une propriété de compatibilité entre deux CPS. Des CPS sont dits compatibles s'ils peuvent mener à des sous-configurations liables entre elles (Propriété 5.5.3). Pour vérifier la compatibilité de deux CPS, il doit exister une association entre les concepts référencés par les deux CPS et leurs états de configuration doivent être compatibles selon cette association (voir Propriété 5.4.12).

Propriété 6.2.1 (Compatibilité de deux CPS) :
Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine.
Soit deux CPS $\mathcal{CPS}_i = \langle \mathcal{D}_i, \mathcal{S}_i, SA_i, \mathcal{SC}_i \rangle$ et $\mathcal{CPS}_j = \langle \mathcal{D}_j, \mathcal{S}_j, SA_j, \mathcal{SC}_j \rangle$.
$\mathcal{CPS}_i$ et $\mathcal{CPS}_j$ sont compatibles si et seulement si :
$\exists\ \mathcal{A}_{ij} \in SA$ tel que $\mathcal{A}_{ij} \rightsquigarrow \{\mathcal{D}_i, \mathcal{D}_j\}$ et $(\mathcal{S}_i, \mathcal{S}_j) \rightsquigarrow \mathcal{A}_{ij}$.
On note la compatibilité des deux CPS : $\mathcal{CPS}_i \rightleftharpoons \mathcal{CPS}_j$.

Nous définissons l'opérateur d'*inclusion* pour deux CPS, comme l'opérateur d'inclusion appliqué sur les états de configuration des CPS.

Propriété 6.2.2 (Inclusion de deux CPS) :
Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine. Soit deux CPS $\mathcal{CPS}_i = \langle \mathcal{D}_i, \mathcal{S}_i, SA_i, \mathcal{SC}_i \rangle$ et $\mathcal{CPS}_j = \langle \mathcal{D}_j, \mathcal{S}_j, SA_j, \mathcal{SC}_j \rangle$.
On dit que $\mathcal{CPS}_i$ est inclus dans $\mathcal{CPS}_j$ si $\mathcal{D}_i = \mathcal{D}_j$ et $\mathcal{S}_i \subseteq \mathcal{S}_j$.
On note alors : $\mathcal{CPS}_i \subseteq \mathcal{CPS}_j$.

Nous définissons également l'opération d'*union* de deux CPS comme l'union des états de configuration et des listes d'actions des deux CPS. L'union de deux CPS n'est autorisée que si l'un des deux CPS peut être inclus dans l'autre selon l'opérateur d'inclusion [1].

Propriété 6.2.3 (Union de deux CPS) :
Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine. Soit deux CPS $\mathcal{CPS}_i = \langle \mathcal{D}_i, \mathcal{S}_i, SA_i, \mathcal{SC}_i \rangle$ et $\mathcal{CPS}_j = \langle \mathcal{D}_j, \mathcal{S}_j, SA_j, \mathcal{SC}_j \rangle$.
L'union de $\mathcal{CPS}_i$ et $\mathcal{CPS}_j$ est possible si et seulement si $\mathcal{CPS}_i \subseteq \mathcal{CPS}_j$ ou $\mathcal{CPS}_j \subseteq \mathcal{CPS}_i$.
En considérant $\mathcal{CPS}_j = \langle \mathcal{D}_i, \mathcal{S}_j, SA_j, \varnothing \rangle$, on a alors : $\mathcal{CPS}_i \cup \mathcal{CPS}_j = \mathcal{CPS}_k$ avec $\mathcal{CPS}_k = \langle \mathcal{D}_i, \mathcal{S}_k, SA_k, \mathcal{SC}_i \rangle$ et $\mathcal{S}_k = \mathcal{S}_i \cup \mathcal{S}_j$ et $SA_k = SA_i \cup SA_j$.

Par ailleurs nous pouvons définir la *complexité* d'un CPS par rapport à la complexité de son état de configuration (voir Propriété 5.3.7).

1. Il est important de noter que deux CPS qui référenceraient des sous-configurations distinctes auraient nécessairement des états de configurations qui ne peuvent être inclus l'un dans l'autre : il est donc impossible de réaliser l'union de deux CPS référençant des sous-configurations distinctes.

Propriété 6.2.4 (Complexité d'un CPS) :
Soit $\mathcal{CPS} = \langle \mathcal{D}, \mathcal{S}, SA, \mathcal{SC} \rangle$ un CPS avec son état de configuration $\mathcal{S}$.

Nous notons la complexité de ce CPS : $\theta(\mathcal{CPS})$ et la définissons par rapport à la complexité de son état de configuration $\mathcal{S}$: $\theta(\mathcal{CPS}) = \theta(\mathcal{S})$.

Afin de simplifier les illustrations nous ne représenterons jamais un CPS : nous utiliserons directement l'état de configuration qu'il représente dans nos exemples.

6.3 Des contextes de configuration

Nous définissons la notion de **contexte** de manière similaire à la notion de portée de variable en programmation impérative. En effet, une variable dans ces types de langage peut être déclarée de manière *globale* pour tout le programme, ou de manière *locale* au niveau d'une fonction ou d'un bloc. Si la variable est déclarée de manière globale, elle sera visible de n'importe où dans le programme. En revanche, si elle est déclarée au sein d'un bloc ou d'une fonction, elle ne sera visible qu'au sein de ce bloc ou de cette fonction.

Nous souhaitons reproduire le même comportement pour les choix effectués par l'utilisateur au sein de la LPL. Certains choix doivent avoir un impact global parce qu'ils contraignent l'ensemble des choix indépendants du contexte – de par la compatibilité entre les FM exprimée dans les fonctions de restriction. D'autres choix en revanche ne vont avoir qu'un impact local sur les voisins immédiats de la configuration en cours.

Exemple 20 : Contexte de sélection

Un gérant d'immeuble peut décider de réaliser un *Immeuble* disposant d'un raccordement à la *Fibre optique* : tous les *Appartements* auront alors la possibilité d'être raccordés à la *Fibre*. Ce choix est global pour l'*immeuble*.

En revanche, un propriétaire d'*Appartement* peut souhaiter de ne pas utiliser la *Fibre* mais d'avoir un *chauffage central* pour son appartement.

Ces choix sont locaux à l'*appartement*.

Nous définissons les contextes comme un ensemble de CPS, chacun étant lié à un concept différent.

Définition 6.3.1 (Contexte) :
Nous définissons un contexte $\mathcal{C}$ comme un unique élément $\langle SCPS \rangle$ où $SCPS = \{\mathcal{CPS}_0, ..., \mathcal{CPS}_n\}$ est un ensemble de CPS tels que : $\forall\ \mathcal{CPS}_i \in SCPS$, avec $\mathcal{CPS}_i = \langle \mathcal{D}_i, \mathcal{S}_i, SA_i, \mathcal{SC}_i \rangle$, $\nexists\ \mathcal{CPS}_j = \langle \mathcal{CPS}_j, \mathcal{S}_j, SA_j, \mathcal{SC}_j \rangle \in SCPS$ tel que $\mathcal{D}_i = \mathcal{D}_j$.

Les contextes sont une des pierres angulaires de la cohérence des choix réalisés par l'utilisateur et de la réalisation des liens entre les configurations. Ils ne peuvent donc contenir que des CPS compatibles entre eux selon les règles de restriction définies dans le modèle du domaine (voir section 5.4).

Propriété 6.3.1 (Cohérence d'un contexte) :
Soit $\mathcal{C} = \langle SCPS \rangle$ un contexte quelconque.
$\forall (\mathcal{CPS}_i, \mathcal{CPS}_j) \in SCPS^2$, alors $\mathcal{CPS}_i \rightleftharpoons \mathcal{CPS}_j$.

Nous distinguons les **contextes locaux** et les **contextes globaux**. Un contexte global contient un CPS pour chacun des concepts du modèle du domaine. Un contexte local en revanche, ne contient que les CPS des concepts ayant une multiplicité avec une limite supérieure non bornée.

Propriété 6.3.2 (Contexte global et local) :
Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine. Soit $\mathcal{C} = \langle SCPS \rangle$ un contexte associé à ce modèle.
Le contexte $\mathcal{C}$ est *global* si $SCPS = \{\mathcal{CPS}_i | \forall\ \mathcal{D}_i \in SC, \mathcal{CPS}_i = \langle \mathcal{D}_i, \mathcal{S}_i, SA_i \rangle\}$.
Le contexte $\mathcal{C}$ est *local* si $SCPS = \{\mathcal{CPS}_i | \forall\ \mathcal{D}_i = \langle id_i, \mathcal{V}_i, \mathcal{U}_i, SA_i \rangle \in SC$ tel que $\mathcal{U}_i = \langle L_i^-, * \rangle, \mathcal{CPS}_i = \langle \mathcal{D}_i, \mathcal{S}_i, SA_i \rangle\}$.

Nous définissons également la complexité d'un contexte, en relation avec la complexité de chacun des CPS qu'il contient (voir Propriété 6.2.4).

Propriété 6.3.3 (Complexité d'un contexte) :
Soit $\mathcal{C} = \langle SCPS \rangle$ un contexte avec $SCPS = \{\mathcal{CPS}_0, ..., \mathcal{CPS}_n\}$.
Nous notons la complexité de ce contexte $\theta(\mathcal{C})$ et nous définissons cette complexité comme la somme de la complexité des CPS qui composent le contexte : $\theta(\mathcal{C}) = \sum_{i=0}^{n} \theta(\mathcal{CPS}_i)$.

Enfin nous disposons d'une entité faisant le lien, durant le processus de configuration, entre le modèle du domaine, les différents contextes de configuration et la configuration composite : **un gestionnaire de configuration composite** (ou **GCC**).

Définition 6.3.2 (Gestionnaire de Configuration Composite) :
Nous définissons un gestionnaire de configuration composite $\mathcal{GCC}$ comme un quintuplet $\langle \mathcal{M}, GC, SLC, \mathcal{CC}, \mathcal{H} \rangle$ où $\mathcal{M}$ représente le modèle du domaine qui est en cours de configuration, GC est un contexte global de configuration, $SLC = \{\mathcal{C}_0, ..., \mathcal{C}_n\}$ est l'ensemble des contextes locaux, $\mathcal{CC}$ la configuration composite en cours de création et $\mathcal{H}$ représente l'historique de configuration que nous définirons dans la suite (voir Définition 6.5.4) qui a mené à cette configuration composite.

Un GCC est défini comme cohérent si :
— chacun de ses contextes est cohérent ;
— les CPS de tous les contextes locaux sont forcément compatibles avec les CPS du contexte global pour lesquels il existe une association ;
— la configuration composite qu'il réalise est cohérente.

Propriété 6.3.4 (Cohérence d'un GCC) :
Soit $\mathcal{GCC} = \langle \mathcal{M}, GC, SLC, \mathcal{CC}, \mathcal{H} \rangle$.
Soit $\mathcal{M} = \langle SC, SA \rangle$.

$\mathcal{GCC}$ est *cohérent* si et seulement si :
- $\forall\, \mathcal{C} \in SLC \cup \{GC\}$, $\mathcal{C}$ est un contexte cohérent (voir Propriété 6.3.1) ;
- $\forall\, \mathcal{A}_{ij} \in SA$ avec $\mathcal{A}_{ij} \rightsquigarrow \{\mathcal{D}_i, \mathcal{D}_j\}$, $\forall\, \mathcal{C} \in SLC$, $\forall\, \mathcal{CPS}_i \in \mathcal{C}$ tel que $\mathcal{CPS}_i$ est relatif au concept $\mathcal{D}_i$, alors $\exists \mathcal{CPS}_j \in GC$ relatif au concept $\mathcal{D}_j$ tel que $\mathcal{CPS}_i \rightleftharpoons \mathcal{CPS}_j$;
- $\mathcal{CC}$ est une configuration composite cohérente (voir Propriété 5.5.4).

Nous définissons la complexité d'un GCC par rapport à la complexité de chacun des contextes qu'il contient (voir Propriété 6.3.3).

Propriété 6.3.5 (Complexité d'un GCC) :
Soit $\mathcal{GCC} = \langle \mathcal{M}, GC, SLC, \mathcal{CC}, \mathcal{H} \rangle$ avec GC un contexte global et $SLC = \{\mathcal{C}_0, ..., \mathcal{C}_n\}$ un ensemble de contexte locaux.

Nous notons la complexité de ce GCC $\theta(\mathcal{GCC})$ et la définissons par la somme de la complexité de chacun des contextes qu'il contient : $\theta(\mathcal{GCC}) = \theta(GC) + \sum_{i=0}^{n} \theta(\mathcal{C}_i)$.

Le GCC reflète l'état global du processus de configuration. Il doit donc être en permanence cohérent par rapport aux actions effectuées par l'utilisateur. Un algorithme de propagation traversant les différents contextes nous permet d'assurer cette cohérence.

6.4 Algorithme de Propagation

Les actions de configurations ont des conséquences différentes en fonction des états de configuration et des contextes dans lesquels elles s'appliquent. Ainsi, nous définissons un **algorithme de propagation** des restrictions qui va nous permettre de garantir à tout instant la cohérence de l'ensemble des contextes (voir Propriété 6.3.1) et donc plus largement la cohérence du GCC.

6.4.1 Description de l'algorithme

Notre algorithme est récursif. Il prend comme arguments un CPS dans lequel une action sur un FM vient éventuellement d'être réalisée, son contexte associé ainsi que leur GCC. L'algorithme ne retourne aucun résultat mais modifie l'état du GCC en fonction des actions réalisées.

L'algorithme se déroule en deux étapes principales :

1. il détermine et applique les actions à effectuer sur les autres CPS, d'après les associations du modèle du domaine et construit une liste des CPS cibles impactés par une action ;
2. puis, pour tous les CPS qui ont été impactés, l'algorithme est relancé en prenant chaque CPS impacté comme point de départ.

Nous décrivons dans la suite les principales fonctions de notre algorithme de propagation.

Algorithme 1 propagate(inout cps :CPS, inout cont :Context, inout gcc :GCC) :void

```
1 : setOfCPS := restriction(cps, cont, gcc)
2 : recursePropagation(setOfCPS, cont, gcc)
```

Lancement de la propagation L'Algorithme 1 présenté correspond à la fonction principale de propagation. Il prend en argument un CPS, le contexte qui lui est associé ainsi que le GCC dans lequel le processus de configuration se déroule. Il effectue tout d'abord une opération de *restriction* que nous détaillons dans la suite, par laquelle il obtient une liste de CPS impactés. A partir de cette liste, il effectue la propagation de manière récursive.

Algorithme 2 restriction(inout cps :CPS, inout cont :Context, inout gcc :GCC) :Set<CPS>

```
1 : result := ∅
2 : conceptSrc := cps.getConcept()
3 : assoSrc := conceptSrc.getAssociations()
4 : for all asso ∈ assoSrc do
5 :     conceptTarget := assoSrc.getOtherEnd(conceptSrc)
6 :     multTarget = conceptTarget.getMultiplicity()
7 :     if multTarget.upperBound() = 1 then
8 :         globalCtx := gcc.getGlobalContext()
9 :         cpsTarget := globalCtx.getCPSofConcept(conceptTarget)
10 :    else
11 :        cpsTarget := cont.getCPSofConcept(conceptTarget)
12 :    setOfActionsToDo := getActions(asso, cps)
13 :    for all action in setOfActionsToDo do
14 :        if action ∈ cpsTarget.getActionsDone() then
15 :            result.add(cpsTarget)
16 :            action.apply(cpsTarget)
17 : return result
```

Application des restrictions La fonction *restriction* présentée dans l'Algorithme 2 a trois rôles : elle récupère les CPS liés au CPS d'entrée en parcourant les associations, elle leur applique les actions données par la fonction *getActions* détaillée par la suite et elle construit une liste de CPS impactés qu'elle retourne.

Les trois premières lignes de l'algorithme permettent respectivement de créer une liste de résultat vide, de récupérer le concept lié au CPS passé en paramètre et de récupérer l'ensemble des associations dans lequel le concept est impliqué.

La suite de l'algorithme se fait en considérant chacune des associations ainsi récupérée (ligne 4). La ligne 5 de l'algorithme permet de récupérer le deuxième concept de l'association considérée.

Les lignes de 6 à 11 permettent de récupérer un CPS lié au second concept de l'association, en fonction de la multiplicité du concept. En effet, si le concept a une multiplicité bornée, alors il n'existe pas de CPS qui le représente dans les contextes locaux (voir Propriété 6.3.2), on récupère donc le CPS à partir du contexte global. En revanche, si la multiplicité du concept est non bornée, il existe un CPS qui le représente dans *tous* les contextes : on récupère donc le CPS dans le même contexte que le CPS d'entrée de la fonction.

Nous récupérons ensuite à la ligne 12 la liste des actions à effectuer sur le CPS, grâce à une fonction que nous détaillons par la suite.

Puis nous itérons sur chacune des actions en vérifiant à la ligne 14 si l'action a déjà été réalisée ou non au sein du CPS. Cela nous permet dans la ligne suivante d'ajouter le CPS à la liste des CPS impactés si l'action n'a pas encore été réalisée. Nous appliquons alors l'action sur le CPS considéré (lignes 15 et 16). Il est important de noter que l'application de l'action

sur un CPS modifie dynamiquement la configuration du FM grâce au raisonnement dynamique (voir section 6.2) : des features sélectionnées et exclues sont donc automatiquement réifiées au niveau de l'état de configuration du CPS.

La fonction se termine en retournant la liste résultante constituée de l'ensemble des CPS impactés.

Algorithme 3 getActions(in asso :Association, in cps :CPS) :Set<Action>

```
1 : result := ∅
2 : CS := cps.getState()
3 : for all rf ∈ asso.getRestrictionFunctions() do
4 :     if rf.getSourceConcept() = cps.getConcept() then
5 :         for all rule ∈ rf.getRules() do
6 :             ruleCS = rule.getState()
7 :             if ruleCS ⊆ CS then
8 :                 result.add(rule.getAction())
9 : return result
```

Récupération des actions La fonction *getActions* présentée dans l'Algorithme 3 est utilisée dans la fonction *restriction* présentée dans l'Algorithme 2 afin de récupérer l'ensemble des actions à appliquer à un CPS.

Cette fonction prend deux arguments en paramètres : l'association à considérer pour l'application des fonctions de restriction et le CPS déterminant l'état de configuration utilisé pour connaître les règles de restrictions à appliquer.

La fonction permet ainsi de récupérer une liste d'actions à appliquer sur un CPS cible, à partir de l'état d'un CPS et d'une association donnés.

La fonction commence donc par définir dans les premières lignes un résultat vide et à récupérer l'état de configuration du CPS donné en paramètre. L'essentiel de l'algorithme consiste ensuite, de la ligne 3 à la ligne 8 à itérer sur l'ensemble des fonctions de restriction puis sur l'ensemble des règles de restriction. Un test est effectué à la ligne 4 afin de vérifier que la fonction de restriction considérée est orientée correctement par rapport au CPS donné.

L'algorithme récupère ensuite l'état de configuration de chacune des règles de la fonction de restriction considérée et teste à la ligne 7 si cet état de configuration peut être inclus dans l'état de configuration du CPS donné afin de savoir si la règle peut être appliquée (voir Définition 5.4.2). Si c'est effectivement le cas, alors la règle et donc l'action peut s'appliquer : l'action est donc ajoutée dans la liste du résultat.

L'algorithme se termine en retournant la liste résultante.

Application récursive de la propagation La fonction *recursePropagation* présentée dans l'Algorithme 4 permet ensuite d'appliquer la propagation de manière récursive. Cette fonction est appelée dans la ligne 2 de l'Algorithme 1.

Cette fonction prend en arguments la liste des CPS qui été calculée lors de l'application de la propagation dans la fonction *restriction* (voir Algorithme 2), le contexte à partir duquel elle a été lancée, ainsi que le GCC dans lequel se déroule le processus de configuration.

La fonction itère sur chacun des CPS de la liste (ligne 1) et relance la fonction de propagation en fonction de la multiplicité du concept auquel se réfère le CPS. Si le CPS donné en argument fait référence à un concept avec une multiplicité non bornée *et* que son contexte est un contexte local, alors la propagation est relancée directement à partir du CPS et de son contexte (ligne 14).

Algorithme 4 recursePropagation(inout setOfCPS :Set<CPS>, inout cont :Context,inout gcc :GCC) :void

```
1 : for all cps ∈ setOfCPS do
2 :   conceptTarget := cps.getConcept()
3 :   multTgt := conceptTarget.getMultiplicity()
4 :   globalCtx := gcc.getGlobalContext()
5 :   if (multTgt.upperBound() = 1) or (cont = globalCtx) then
6 :     propagate(cps, globalCtx)
7 :     if not multTgt.upperBound() = 1 then
8 :       setLocalCtx := gcc.getLocalContexts()
9 :       for all locCtx ∈ setLocalCtx do
10 :        localCPS := locCtx.getCPSofConcept(conceptTarget)
11 :        merge(localCPS, cps)
12 :        propagate(localCPS, locCtx)
13 :  else
14 :    propagate(cps, cont)
```

En revanche, si le CPS donné en argument est lié à un concept dont la multiplicité est bornée *ou* si le CPS appartient au contexte global, alors la propagation doit être lancé à partir du contexte global (ligne 6).

Enfin, il faut considérer le cas particulier d'un CPS qui est lié à un concept dont la multiplicité est non bornée, mais qui a été modifié au sein du contexte global. Cela arrive dans le cas où la propagation a été lancée à partir d'un concept borné dans le contexte global et qu'une règle a été appliquée sur un concept possédant une multiplicité non bornée. Par exemple si l'on a sélectionné une feature de l'*Immeuble* qui implique une sélection dans le FM *Appartement*.

En effet, dans ce cas chaque contexte local possède aussi un CPS lié au même concept et les modifications apportées au sein du contexte global doivent être reflétées au sein des contextes locaux.

Dans ce cas l'algorithme itère sur chacun des contextes locaux (ligne 9) afin, tout d'abord, d'appliquer aux CPS locaux les mêmes modifications que celles qui ont été appliquées au CPS global (ligne 11) puis, ensuite, de relancer la propagation à partir de ces CPS locaux nouvellement modifiés (ligne 12). La fonction applique donc une propagation en profondeur d'abord.

6.4.2 Propriétés de l'algorithme

Soit une LPL représentée par un modèle $\mathcal{M}$ cohérent (voir section 5.6). En considérant $\mathcal{GCC}$ un GCC *cohérent* s'appliquant sur $\mathcal{M}$ et $\mathcal{GCC}'$ obtenu après l'application de l'algorithme de propagation à partir d'un contexte et d'un CPS quelconque de $\mathcal{GCC}$. Alors nous pouvons garantir plusieurs propriétés sur l'algorithme de propagation à partir de $\mathcal{GCC}'$:

1. le GCC $\mathcal{GCC}'$ est toujours cohérent (voir Propriété 6.3.4) suite à la propagation ;
2. la propagation va toujours réduire ou conserver la complexité du GCC : la complexité de $\mathcal{GCC}'$ est nécessairement inférieure ou égale à la complexité de $\mathcal{GCC}$;
3. la propagation est stable : l'application de l'algorithme de propagation sur $\mathcal{GCC}'$ a nécessairement pour résultat $\mathcal{GCC}'$ quels que soient le contexte et le CPS de lancement de la propagation, aucune action supplémentaire n'ayant été réalisée sur $\mathcal{GCC}'$;

4. la propagation termine toujours.

6.4.2.1 Cohérence du GCC

Soit $\mathcal{C}'$ un contexte de $\mathcal{GCC}'$ et $\mathcal{A}_{ij}$ une association de $\mathcal{M}$ reliant $\mathcal{D}_i$ et $\mathcal{D}_j$. Soit $\mathcal{CPS}_i$ et $\mathcal{CPS}_j$ appartenant à $\mathcal{C}'$ et référençant respectivement $\mathcal{D}_i$ et $\mathcal{D}_j$.

Nous démontrons cette propriété en considérant qu'il existe $\mathcal{CPS}_i$ et $\mathcal{CPS}_j$ qui ne sont pas compatibles (voir Propriété 6.2.1) et nous montrons que cela est impossible.

Démonstration. Soit $\mathcal{S}_i$ et $\mathcal{S}_j$ les états de configurations respectifs de $\mathcal{CPS}_i$ et $\mathcal{CPS}_j$. Si $\mathcal{CPS}_i$ et $\mathcal{CPS}_j$ ne sont pas compatibles, alors $\exists \mathcal{RF}_{i\to j}$ une règle de restriction appartenant à $\mathcal{A}_{ij}$ telle que $applyRF(\mathcal{RF}_{i\to j}, \mathcal{S}_i, \mathcal{S}_j) \to \mathcal{S}'_j$ avec $\mathcal{S}'_j \neq \mathcal{S}_j$ (voir Propriété 5.4.11).

Il existe deux possibilités pour que la fonction de restriction $\mathcal{RF}_{i\to j}$ n'ait pas été appliquée :

— soit la propagation n'a pas été lancée à partir de $\mathcal{CPS}_i$ car celui-ci n'a pas été impacté par une action, mais dans ce cas $\mathcal{CPS}_i$ et $\mathcal{CPS}_j$ étaient compatibles puisque le GCC $\mathcal{GCC}$ était cohérent ;

— soit l'action permettant d'obtenir $\mathcal{S}'_j$ n'a pas été appliquée car elle avait été précédemment appliquée, et dans ce cas $\mathcal{CPS}_j = \mathcal{CPS}'_j$.

Il est donc impossible que des CPS incompatibles co-existent au sein d'un contexte à la fin d'une propagation : l'algorithme garantit la cohérence du GCC. □

6.4.2.2 Réduction de la complexité

Par définition la complexité du GCC est liée *in fine* à la complexité de chaque état de configuration contenu dans le GCC (voir Propriété 6.3.5, Propriété 6.3.3, Propriété 6.2.4). Or l'application d'une action de configuration ne peut que réduire ou maintenir la complexité d'un état de configuration (voir Propriété 5.4.1). En outre l'algorithme de propagation ne modifie pas le nombre de contextes créés.

En conclusion, l'algorithme de propagation ne peut que réduire ou maintenir la complexité des différents états de configuration contenus dans le GCC. Par conséquent la complexité du GCC est soit stable, soit réduite suite à une propagation.

6.4.2.3 Stabilité

Nous définissons la *stabilité* de notre algorithme de propagation, par le fait que l'application de l'algorithme de propagation sur un GCC qui vient d'être modifié par une première application de la propagation ne fait rien.

Quels que soient $\mathcal{GCC}$ un GCC, $\mathcal{C}$ un contexte de $\mathcal{GCC}$ et $\mathcal{CPS}$ un CPS de $\mathcal{C}$, alors : $propagate(\mathcal{CPS}, \mathcal{C}, \mathcal{GCC}) \to \mathcal{GCC}'$.

Quels que soient $\mathcal{C}'$ un contexte de $\mathcal{GCC}'$ et $\mathcal{CPS}'$ un CPS de $\mathcal{C}'$, alors : $propagate(\mathcal{CPS}', \mathcal{C}', \mathcal{GCC}') \to \mathcal{GCC}''$.

Nous montrons que $\mathcal{GCC}'' \equiv \mathcal{GCC}'$.

Démonstration. Si $\mathcal{GCC}'' \neq \mathcal{GCC}'$, cela signifie qu'il existe une règle de restriction $\mathcal{R}$ applicable à partir d'un des CPS de $\mathcal{GCC}'$ et qu'elle n'a pas été appliquée.

Cela suppose :

1. soit qu'il existe un CPS dont l'état de configuration correspond à l'état d'application de la règle $\mathcal{R}$ mais que la règle $\mathcal{R}$ n'a pas été appliquée ;
2. soit que l'action correspondante à la règle $\mathcal{R}$ n'a pas été exécutée.

Nous montrons que ces deux situations sont impossibles.

1. S'il existe un CPS dont l'état de configuration correspond à l'état d'application d'une règle, alors le CPS a été modifié lors de la propagation par une action qui lui a été appliquée automatiquement, auquel cas le CPS est noté comme "impacté" et la propagation est de nouveau appliquée à partir de ce CPS (voir Algorithme 1).
2. Il n'existe qu'une seule possibilité dans notre algorithme pour qu'une action provenant d'une règle ne soit pas exécutée : il s'agit du cas où l'action a déjà été exécutée auparavant (voir Algorithme 2 ligne 14) soit par l'application de la règle soit par une action utilisateur.

Ainsi, nous avons toujours $\mathcal{GCC}'' = \mathcal{GCC}'$, la propagation est donc stable pour son application. □

6.4.2.4 Terminaison

La terminaison de notre algorithme de propagation est assurée par trois propriétés :
— un CPS n'est considéré que s'il est impacté par une action ;
— deux actions identiques n'ont jamais lieu sur le même CPS et un CPS n'est considéré que s'il est impacté par une action ;
— l'ensemble des actions possibles sur un CPS est limité par le nombre de features du FM configuré au sein du CPS : ainsi, le nombre d'actions possibles décroît à chaque étape de la propagation.

Démonstration. Soit $SLC = \{\mathcal{C}_0, ..., \mathcal{C}_n\}$ un ensemble de contextes locaux et GC un contexte global. Soit $\mathcal{CPS}_i \in \mathcal{C}_i$ un CPS quelconque.

Considérons le lancement de l'algorithme de propagation à partir du CPS $\mathcal{CPS}_i$ et du contexte $\mathcal{C}_i \in SLC$. L'algorithme continue de s'exécuter tant qu'il existe des actions à propager.

Or nous effectuons dans l'Algorithme 2 un test permettant de vérifier si une action a déjà été effectuée ou non avant de l'appliquer (voir ligne 14) et donc de considérer le CPS comme impacté. Il est donc impossible dans un CPS donné d'appliquer plusieurs fois une action identique.

Ainsi l'algorithme peut continuer de s'exécuter tant qu'il existe encore une action réalisable dans un des CPS. Or il a été montré dans la Propriété 5.4.5 que la liste des actions possibles dans un FM est bornée par le nombre de features du FM.

Donc l'algorithme ne peut s'exécuter indéfiniment que si de nouveaux CPS sont créés. Or la définition d'un contexte précise qu'il n'existe qu'un seul CPS par FM dans un contexte donné. Il n'est donc possible de créer de nouveaux CPS qu'en créant de nouveaux contextes.

Or la propagation ne crée jamais de nouveaux contextes et la liste des contextes locaux est finie. En conclusion, il est impossible que l'algorithme de propagation ne s'arrête jamais : il y aura donc toujours une terminaison à notre algorithme de propagation. □

6.5 Un processus de configuration par étapes

Le processus de configuration d'un nouveau produit repose sur les actions demandées par les utilisateurs. Dans l'objectif de garantir un processus de configuration flexible il convient d'être en mesure d'appliquer le plus librement possible ces actions. Cependant, le besoin de conservation d'un processus cohérent requiert de considérer les conditions d'application des actions. Nous définissons donc un formalisme d'actions à deux niveaux constitués d'actions

utilisateur, qui représentent les actions de haut niveau demandées par un utilisateur, et d'actions système qui sont créées et appliquées automatiquement par les actions utilisateur.

Par ailleurs, nul n'étant infaillible et afin, une fois encore, d'améliorer la flexibilité du processus de configuration, il est indispensable que l'utilisateur puisse être en mesure d'annuler une de ses actions. Nous définissons ainsi un algorithme d'annulation d'une action utilisateur et les éléments du formalisme nécessaire à cet algorithme.

6.5.1 Actions système et actions utilisateur

Nous considérons dans notre modèle deux types d'actions : les **actions système** et les **actions utilisateur** [2].

Nous définissons une action système comme une action atomique intervenant directement sur les notions définies dans le formalisme (chapitre 5 et chapitre 6) et appliquée automatiquement par une action utilisateur.

Une action système est définie par un type et un ensemble d'arguments. Chaque action système s'applique sur un GCC qu'elle modifie.

Type d'action	Description
Sélectionner une feature[3]	Ce type d'action sélectionne une feature d'un CPS spécifique.
Exclure une feature[3]	Ce type d'action exclut une feature d'un CPS spécifique.
Ajouter une contrainte[3]	Ce type d'action ajoute une contrainte sur le FM d'un CPS spécifique.
Créer une sous-configuration	Ce type d'action crée une sous-configuration à partir d'un CPS.
Référencer une sous-configuration	Ce type d'action détermine qu'une sous-configuration est référencée par un autre CPS que son CPS d'origine.
Créer un lien	Ce type d'action crée un lien entre deux sous-configurations.
Créer un contexte	Ce type d'action crée un nouveau contexte local.
Supprimer un contexte	Ce type d'action supprime un contexte local.

TABLE 6.2 – Liste des types d'actions système

Définition 6.5.1 (Action système) :
Soit $\mathcal{AS} = \langle type, arguments \rangle$ une action système.

Les arguments d'une action système sont nécessairement des entités du formalisme défini dans les chapitres 5 et 6. Nous définissons la fonction $apply$ d'une action système qui permet d'appliquer l'action à partir de son type et de ses arguments et modifie l'état du GCC ainsi que l'état de l'action.

Soit $\mathcal{GCC}$ un GCC et $\mathcal{AS}$ une action système quelconque applicable à ce GCC, au sens où les arguments de l'action sont des éléments du GCC.

2. Seules les actions qui permettent de configurer un produit sont présentées ici. Certaines actions utilitaires supplémentaires ont été réalisées et sont évoquées dans la description de l'implémentation réalisée section 7.3

3. Il s'agit des actions système correspondant aux actions de configurations de FM définies en sous-section 5.4.1.

On a alors : $apply(\mathcal{A}, \mathcal{GCC}) \rightarrow \mathcal{GCC}'$.

Si une action n'est pas applicable à un GCC, elle ne fait rien. Il est important de noter que les actions de configurations sur les FM (Définition 5.4.1) sont des actions système.

Propriété 6.5.1 (Annuler une action système) :
Nous définissons la fonction $undo$ pour une action système. Cette fonction prend en paramètre une action système et un GCC résultant de l'application de l'action et permet d'annuler celle-ci au sein du GCC.

Soit $\mathcal{AS}$ une action système quelconque et $\mathcal{GCC}$ un GCC sur lequel peut s'appliquer cette action.

On a : $undo(\mathcal{AS}, apply(\mathcal{AS}, \mathcal{GCC})) \rightarrow \mathcal{GCC}$.

Le Tableau 6.2 montre la liste des types d'actions système actuellement supportées.

Le processus de configuration est constitué d'une suite d'actions utilisateur menant à une configuration composite spécifique. Nous définissons une action utilisateur comme une séquence d'actions système nécessaires à la réalisation d'un but de plus haut niveau.

Une action utilisateur est formellement définie par un type et un ensemble d'arguments.

Définition 6.5.2 (Action utilisateur) :
Soit $\mathcal{Q} = \langle type, arguments \rangle$ une action utilisateur.

Nous définissons la fonction $precondition$ qui prend comme argument une fonction utilisateur et un GCC et qui retourne une valeur booléenne `vraie` si une action de ce type et avec ces arguments peut être réalisée sur le GCC et `faux` dans le cas contraire.

Nous définissons la fonction $apply$ qui applique l'action utilisateur sur un GCC et retourne un GCC modifié.

Soit $\mathcal{GCC}$ un GCC quelconque.

Soit $\mathcal{Q}$ une action utilisateur quelconque, telle que $precondition(\mathcal{Q}, \mathcal{GCC}) = vrai$.

Alors $apply(\mathcal{Q}, \mathcal{GCC}) \rightarrow \mathcal{GCC}'$.

Si $precondition(\mathcal{Q}, \mathcal{GCC}) = faux$ alors $apply(\mathcal{Q}, \mathcal{GCC}) \rightarrow \mathcal{GCC}$.

Le Tableau 6.4 représente l'ensemble des types d'actions utilisateur supportés.

6.5.2 Historique de configuration

Nous définissons une **étape de configuration** comme une action utilisateur et l'ensemble des actions système issues de l'application de cette action utilisateur. Nous conservons au sein d'un GCC un **historique de configuration** contenant la liste des étapes de configuration. Cette approche nous permet d'annuler une action utilisateur en garantissant la cohérence du processus de configuration.

Définition 6.5.3 (Etape de configuration) :
Nous définissons une étape de configuration $\mathcal{T}$ comme un couple $\langle \mathcal{Q}, SA \rangle$ avec $\mathcal{Q}$ l'action utilisateur à l'origine de l'étape de configuration et $SA = [\mathcal{AS}_0, ..., \mathcal{AS}_n]$ la liste ordonnée des actions système résultants de l'application de l'action utilisateur.

Type d'action	Description
Initialisation	Ce type d'action initialise l'ensemble du système, en vérifiant la cohérence de la SPL et en créant le contexte global.
Finalisation	Ce type d'action récupère les informations d'une configuration composite valide et terminée afin de l'utiliser, par exemple, lors d'un processus de génération ou de composition.
Sélectionner une feature	Ce type d'action sélectionne une feature d'un concept précis au sein d'un contexte donné. Une propagation est automatiquement lancée à la suite d'une sélection.
Exclure une feature	Ce type d'action exclut une feature d'un concept précis au sein d'un contexte donné. Une propagation est automatiquement lancée à la suite d'une exclusion.
Démarrer une sous-configuration	Ce type d'action commence une nouvelle sous-configuration.
Valider une sous-configuration	Ce type d'action termine une sous-configuration afin de l'ajouter dans la configuration composite.
Lier des sous-configurations	Ce type d'action crée un nouveau lien dans la configuration composite entre deux sous-configurations existantes.

TABLE 6.4 – Liste des types d'actions utilisateur

Nous définissons le concept d'**historique de configuration**, comme l'ensemble ordonné des étapes de configuration réalisées.

Définition 6.5.4 (Historique de configuration) :
Nous définissons un historique de configuration $\mathcal{H} = \langle ST \rangle$ où $ST = [\mathcal{T}_0, ..., \mathcal{T}_n]$ représente l'ensemble ordonné des étapes de configuration.

L'application d'une action utilisateur modifie le GCC en effectuant des changements sur la configuration composite ou les contextes, mais aussi en ajoutant automatiquement à l'historique une étape de configuration.

Propriété 6.5.2 (Application d'une action utilisateur) :
Nous détaillons la définition de la fonction $apply$ sur une action utilisateur qui consiste donc à :

- créer une nouvelle étape de configuration au sein de l'historique du GCC ;
- créer une liste d'actions système au sein de cette étape de configuration ;
- appliquer chacune des actions système dans l'ordre sur le GCC.

Soit $\mathcal{Q}$ une action utilisateur et $\mathcal{GCC} = \langle \mathcal{M}, GC, SLC, \mathcal{CC}, \mathcal{H} \rangle$ un GCC tel que $precondition(\mathcal{Q}, \mathcal{GCC}) = vrai$, alors on a : $apply(\mathcal{Q}, \mathcal{GCC}) \rightarrow \mathcal{GCC}'$ avec $\mathcal{GCC}' = \langle \mathcal{M}, GC', SLC', \mathcal{CC}', \mathcal{H}' \rangle$ où

- $\mathcal{H}' = \langle ST' \rangle$ avec $ST' = [\mathcal{T}_0, ..., \mathcal{T}_{n+1}]$ en considérant $\mathcal{H} = \langle ST \rangle$ avec $ST = [\mathcal{T}_0, ..., \mathcal{T}_n]$ et $\mathcal{T}_{n+1} = \langle \mathcal{Q}, SA \rangle$ avec $SA = [\mathcal{AS}_0, ..., \mathcal{AS}_n]$;

— et $apply(\mathcal{AS}_n, apply(\mathcal{AS}_{n-1}, ..., apply(\mathcal{AS}_0, \mathcal{GCC})...)) \rightarrow \mathcal{GCC}''$
avec $\mathcal{GCC}'' = \langle \mathcal{M}, GC', SLC', \mathcal{CC}', \mathcal{H} \rangle$.

De la même façon qu'il est possible d'annuler une action système, nous définissons une fonction $undo$ pour annuler une action utilisateur. L'annulation d'une action utilisateur consiste à annuler l'étape de configuration créée par l'action utilisateur.

Propriété 6.5.3 (Annulation d'une étape de configuration) :
Nous définissons la fonction $undo$ pour une étape de configuration comme l'annulation de toutes les actions système réalisées dans l'ordre inverse de leur exécution et la suppression de cette étape de configuration au sein de l'historique.

La fonction $undo$ prend comme argument une étape de configuration ainsi que le GCC la contenant. La fonction $undo$ retourne un GCC dans lequel les actions système ont été annulées et l'étape de configuration supprimée.

Soit $\mathcal{Q}$ une action utilisateur et $\mathcal{GCC}$ un GCC tel que $precondition(\mathcal{Q}, \mathcal{GCC}) = vrai$.

Soit $\mathcal{GCC}'$ le résultat de la fonction $apply(\mathcal{Q}, \mathcal{GCC})$ et $\mathcal{T}$ l'étape de configuration résultante contenue dans $\mathcal{GCC}'$.

Alors $undo(\mathcal{T}, \mathcal{GCC}') \rightarrow \mathcal{GCC}$.

6.5.3 Algorithme d'annulation d'une action utilisateur

La fonction d'annulation définie sur les étapes de configuration n'est pas suffisante pour garantir la cohérence du GCC lors de l'annulation de n'importe quelle étape. Nous proposons donc un algorithme permettant d'annuler n'importe quelle étape de configuration tout en assurant la cohérence du GCC.

Algorithme 5 undoAction(inout step :Step, inout gcc :GCC) :void

```
1 : actionsUndone  := searchAndUndoActions(gcc, step)
2 : actionsToRedo  := reverseOrder(actionsUndone)
3 : redoActions(actionsToRedo, gcc)
```

Le fonctionnement global de l'algorithme est présenté dans la fonction *undoAction* (Algorithme 5). Cette fonction prend en paramètre d'entrée une étape de configuration à annuler et un GCC contenant cette étape de configuration. Elle ne retourne rien mais modifie le GCC donné en entrée.

Algorithme 6 searchAndUndoActions(inout gcc :GCC, inout step :Step) :List<Action>

```
1 : actionUndone  := new List()
2 : repeat
3 :     history  := gcc.history()
4 :     hStep  := history.lastStep()
5 :     undo(hStep, gcc)
6 :     if hStep ≠ step then
7 :         actionUndone.add(hStep.userAction())
8 : until hStep = step
9 : return  actionUndone
```

En premier lieu, cette fonction va appeler la fonction *searchAndUndoActions* (Algorithme 6) qui va parcourir la liste des étapes de configuration à partir de la fin et les annuler une par une jusqu'à trouver l'étape de configuration donnée en entrée. Cette fonction retourne la liste des actions utilisateur qui ont été annulées.

Dans la deuxième ligne de la fonction *undoActions*, la liste des actions annulées est inversée afin de pouvoir considérer comme premier élément la dernière action annulée.

Enfin, une dernière fonction *redoActions* (Algorithme 7) est appelée prenant comme argument la liste inversée des actions annulées et le GCC. Cette fonction parcourt la liste des actions annulées, vérifie que les préconditions sont toujours respectées et, si c'est le cas les applique à nouveau. Si les préconditions ne sont pas respectées, un message est retourné.

Algorithme 7 redoActions(in listOfRedoActions :List<Action>, inout gcc :GCC) :void

```
1 : for all actionToRedo ∈ listOfRedoActions do
2 :   if actionToRedo.precondition(gcc) then
3 :     GCC := apply(actionToRedo, gcc)
4 :   else
5 :     print "Action actionToRedo.description() cannot be applied
        back"
```

6.5.3.1 Terminaison

La terminaison de l'algorithme est assurée par le fait qu'il parcourt une liste d'étapes de configuration dans laquelle aucune nouvelle étape ne peut être ajoutée durant l'algorithme. La taille de la liste étant ainsi bornée et les opérations sur les étapes de configuration étant également bornées par la taille des listes d'actions système, l'algorithme se termine toujours.

6.5.3.2 Cohérence

L'algorithme nous garantit que la cohérence du processus de configuration est toujours respectée après son exécution. Cette propriété est vérifiée tout d'abord car l'algorithme annule un ensemble d'étapes de configuration dans l'ordre inverse de leur application, en respectant la Propriété 6.5.3 qui vérifie la cohérence du processus de configuration.

Par ailleurs, l'exécution des actions utilisateur qui s'ensuit dans la deuxième partie de l'algorithme n'est effectuée que si les préconditions des actions sont respectées. Nous montrons dans la section suivante comment les actions utilisateur garantissent de vérifier la cohérence du système si leurs préconditions sont respectées.

La cohérence du processus de configuration est ainsi vérifiée par l'algorithme d'annulation d'une action utilisateur.

6.6 Cohérence et flexibilité du processus de configuration

Notre formalisme d'action et les algorithmes de propagation et d'annulation des actions effectuées visent tous trois à garantir la cohérence et la flexibilité du processus de configuration (voir Propriété 6.1.1).

Il convient cependant d'interdire à l'utilisateur des actions qui ne sont pas cohérentes relativement à l'état du GCC.

Nous montrons ainsi dans un premier temps un automate définissant l'ordre autorisé d'exécution des actions utilisateur, et nous définissons la validation de leurs arguments. Nous présentons ensuite comment nos actions utilisateur les plus complexes sont définies afin de garantir les propriétés de cohérence et de flexibilité du processus.

6.6.1 Automate des actions utilisateur et préconditions

La cohérence du processus de configuration est directement issue de la cohérence du GCC (voir Propriété 6.3.4) mais repose également sur les fonctions de *precondition* associées aux actions utilisateur. Ces fonctions sont définies indépendamment pour chaque type d'action. Elles visent à vérifier deux choses :

— est-ce que l'état du GCC permet d'appliquer ce type d'action ?
— est-ce que les arguments de l'action sont compatibles avec l'état du GCC ?

Nous montrons l'automate des actions autorisées durant le processus de configuration, ainsi que le mécanisme de validation des arguments des actions utilisateur.

6.6.1.1 Automate des actions utilisateur

La Figure 6.2 représente cet automate : chaque état correspond à un type d'action utilisateur et chaque transition correspond à une transition autorisée entre deux types d'actions. Il est important de noter que cette figure ne représente pas les conditions qui doivent être vérifiées afin de valider la transition : par exemple, une action de liaison de configurations ne peut être effectuée que si les deux configurations à lier ont été validées au préalable. Nous décrivons ces conditions dans la sous-section suivante.

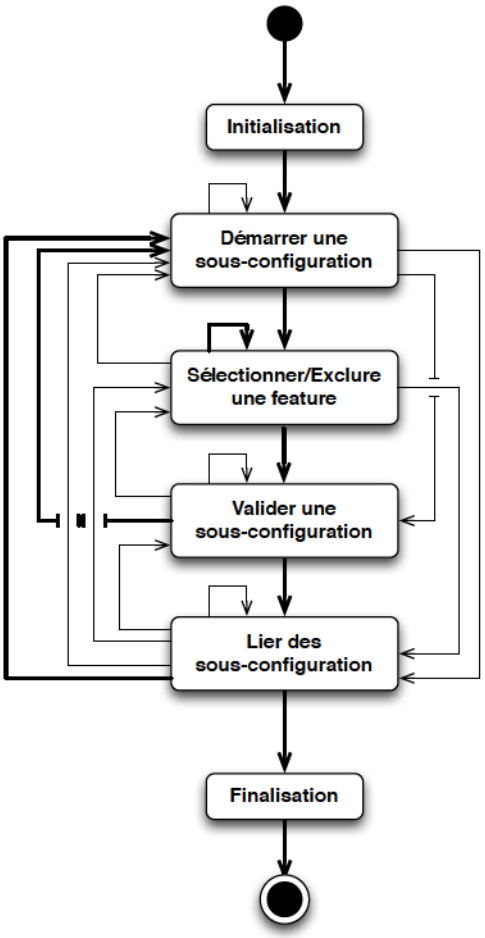


FIGURE 6.2 – Automate des actions utilisateur

Toutes les transitions représentées sont des transitions autorisées au sein de notre processus de configuration si les conditions sont respectées. Ainsi nous pouvons constater que très peu de transitions sont illégales : l'ordre d'application des différentes actions utilisateur est extrêmement libre, ce qui est un gage de la flexibilité du processus. En outre, nous avons volontairement représenté en gras les transitions qui nous ont semblé *le plus souvent* empruntées d'après nos expérimentations. On peut ainsi en déduire qu'un workflow "usuel" de configuration d'un produit commence par une *initialisation*, se poursuit par le *démarrage d'une nouvelle sous-configuration*, enchaîne sur des séries de *sélection/exclusions de features*, pour ensuite permettre à l'utilisateur de *valider la sous-configuration* réalisée. Quand l'utilisateur a défini plusieurs sous-configurations, il va pouvoir créer des *liaisons* entre elles ce qui lui

permettra de *finaliser* sa configuration composite quand le dernier lien aura été créé.

Bien sûr, à tout moment l'utilisateur a la possibilité de sortir de ce workflow : par exemple, il peut se rendre compte au cours de la réalisation d'une sous-configuration qu'il va devoir en créer une autre très similaire et décider d'en démarrer une nouvelle immédiatement, au lieu de finaliser sa sous-configuration en cours.

Nous avons délibérément omis de représenter la possibilité de défaire une action dans notre Figure 6.2 : l'*annulation* de n'importe quelle action utilisateur (voir sous-section 6.5.3) est accessible à n'importe quel moment du workflow de configuration et ne compromet jamais la cohérence du processus.

6.6.1.2 Validation des arguments

Pour chacune des actions utilisateur, les arguments de l'action doivent être validés avant de pouvoir autoriser l'application de l'action. Cette validation est effectuée lors du test de la *precondition* de l'action (voir Définition 6.5.2).

Les arguments des actions référencent toujours des éléments existants qu'il est possible de retrouver à partir du GCC : soit des éléments du modèle du domaine (par exemple, une feature, un concept, ou une association), soit des éléments issus de la configuration (par exemple, un contexte, ou un CPS). La première vérification à faire est donc de s'assurer que les éléments référencés existent bien dans le GCC en cours d'utilisation.

Enfin, pour chacun des types d'action utilisateur, un ensemble de vérifications spécifiques doit également être effectué afin de garantir la cohérence du processus. Les préconditions spécifiques pour les différents types d'actions utilisateur sont décrites dans la suite.

Initialisation Il n'y a pas de précondition spécifique pour ce type d'action utilisateur : celle-ci ne prend en effet aucun argument qu'il conviendrait de vérifier.

Démarrer une nouvelle sous-configuration Ce type d'action utilisateur prend trois arguments : un obligatoire et deux optionnels. L'argument obligatoire est le concept pour lequel l'utilisateur souhaite créer une nouvelle sous-configuration. Le premier argument optionnel est le contexte dans lequel l'utilisateur souhaite la créer. Lorsque cet argument n'est pas renseigné, le contexte considéré est un nouveau contexte local si le concept demandé possède une multiplicité non bornée, sinon il s'agit du contexte global. Enfin le dernier argument qui est également optionnel représente le concept avec lequel l'utilisateur souhaiterait lier sa nouvelle sous-configuration. Plus précisément, cet argument n'est pris en compte que si le premier argument optionnel est donné : il permet d'obtenir à partir du contexte demandé, le CPS du deuxième concept donné et ainsi garantir la création d'une sous-configuration compatible avec ce CPS [4].

Il est nécessaire de vérifier que la multiplicité sera encore respectée une fois la sous-configuration réalisée pour autoriser cette action. Cela signifie que pour un concept possédant une multiplicité bornée, il ne sera possible de démarrer une nouvelle sous-configuration qu'une unique fois, celle-ci étant réalisée dans le contexte global. Il convient donc de vérifier dans ce cas que la sous-configuration n'a pas déjà été validée au sein de la configuration composite.

Par ailleurs, il est possible d'indiquer le contexte dans lequel on souhaite créer la sous-configuration. Là encore il est nécessaire de vérifier que la multiplicité du concept n'est pas contradictoire avec le contexte demandé : si le concept possède une multiplicité bornée, la sous-configuration ne peut pas être créée dans un contexte local. Par ailleurs, si le contexte

4. Des précisions sont données sur ce mécanisme dans la suite de la section : sous-section 6.6.4

donné possède déjà une sous-configuration validée pour le concept demandé, l'action ne peut pas être appliquée non plus.

Enfin, il est possible également d'indiquer un deuxième concept afin que l'utilisateur puisse créer une sous-configuration qui soit compatible avec le CPS de ce concept dans le contexte donné. Dans ce cas, l'utilisateur crée une sous-configuration dans l'optique de la lier à une autre et les multiplicités ont un rôle important à jouer : nous considérons ainsi qu'il est nécessaire qu'il existe une association entre les deux concepts donnés pour que l'action soit validée. Par ailleurs, si une sous-configuration a déjà été validée pour le deuxième concept, il est nécessaire de vérifier que la multiplicité de l'association sera toujours respectée après la création d'un lien.

Sélectionner / Exclure une feature Ces deux types d'actions prennent les mêmes deux arguments obligatoires. Le premier argument est le CPS sur lequel l'utilisateur est en train de travailler pour réaliser une sous-configuration. Le second argument est le nom d'une feature à sélectionner ou exclure, selon le type d'action.

La première vérification à réaliser consiste à s'assurer que la feature appartient bien au FM référencé par le CPS. Il est ensuite nécessaire de vérifier que cette feature n'a pas déjà été sélectionnée ou exclue afin de valider l'action.

Valider une sous-configuration Ce type d'action utilisateur prend un unique argument obligatoire : le CPS dont on souhaite valider la sous-configuration. Il est nécessaire de vérifier que l'état de configuration du CPS est complet et cohérent avant de valider l'action (voir Propriété 5.3.2).

Lier des sous-configurations Ce type d'action prend deux arguments obligatoires : les deux sous-configurations à lier.

Il est tout d'abord nécessaire de vérifier qu'il existe bien une association entre les concepts référencés par les deux sous-configurations. Il faut ensuite déterminer si les deux sous-configurations ne sont pas déjà liées ensemble. Une troisième vérification doit s'assurer que les deux sous-configurations sont compatibles entre elles. Enfin, il faut vérifier pour chacune des sous-configurations qu'elle respecte bien la multiplicité de l'association afin de pouvoir valider l'action.

Finalisation Ce type d'action prend un unique argument obligatoire : la configuration composite réalisée.

La finalisation ne peut être validée que si la configuration composite est cohérente et terminée (voir Propriété 5.5.5).

Nous donnons dans les sous-sections qui suivent des précisions sur le mécanisme de ces différentes actions.

6.6.2 Sélection et exclusions de features

Ces deux types d'actions utilisateur se comportent de la même façon : elles ont pour rôle d'effectuer une action de configuration sur le FM référencé par le CPS donné en argument de l'action.

Cependant, l'action sur le FM seule n'est pas suffisante pour garantir la cohérence du processus de configuration : les fonctions de restriction doivent en effet être appliquées immédiatement

avant toute nouvelle action. Les actions utilisateur de sélection et d'exclusion ont donc la responsabilité de déclencher le lancement de l'algorithme de propagation que nous avons défini précédemment (voir section 6.4). Ainsi, à chaque nouveau choix de configuration réalisé, nous garantissons que le GCC reste cohérent d'après la Propriété 6.3.4 ce qui est un gage de la cohérence du processus de configuration mais aussi de sa flexibilité.

6.6.3 Liaison des sous-configurations

L'action utilisateur *liaison des sous-configurations* pose une grande problématique de cohérence et de flexibilité du processus de configuration. En effet, outre son action de création d'un lien entre les sous-configurations, ce type d'action utilisateur est responsable de certaines opérations sur les contextes.

Selon la multiplicité des associations, des sous-configurations liées doivent ou non être encore libres de pouvoir être liées avec d'autres éléments. Ainsi, il doit encore être possible de créer des sous-configurations compatibles pour certains éléments liés, sans quoi nous contreviendrions à la propriété de flexibilité du processus de configuration.

La propriété de cohérence des contextes garantit que des CPS appartenant au même contexte sont compatibles (voir Propriété 6.3.1). Ainsi des sous-configurations issues d'un même contexte peuvent nécessairement être liées s'il existe une association. Cependant, l'utilisateur doit pouvoir également lier des sous-configurations réalisées dans des contextes différents.

Il est donc indispensable d'avoir un mécanisme permettant *(i)* de vérifier que les deux sous-configurations issus de contextes différents sont compatibles, *(ii)* de réaliser des opérations de fusion de contextes afin de réunir au sein d'un même contexte les CPS référençant des sous-configurations à lier, *(iii)* gérer la suppression des contextes devenus inutiles.

Toutes ces opérations sont réalisées au sein de l'action utilisateur permettant de lier des configurations et permettent de s'assurer de la propriété de flexibilité du processus de configuration tout en garantissant sa cohérence.

6.6.3.1 Vérifier la compatibilité de deux sous-configurations de contextes différents

Si les deux sous-configurations sont issues d'un même contexte ou si une des sous-configurations provient du contexte global, elles sont nécessairement compatibles d'après la propriété de cohérence du GCC (voir Propriété 6.3.4) qui est vérifiée notamment grâce à l'algorithme de propagation. Le lien est alors directement créé.

En revanche, si les sous-configurations à lier sont issues de contextes locaux distincts, il est nécessaire de vérifier leur compatibilité afin de permettre la liaison. Des sous-configurations issues de contextes distincts sont définies comme compatibles pour la liaison si :

— les CPS qu'elles référencent sont compatibles,

— les contextes sont compatibles à la fusion.

La compatibilité des CPS a été définie dans la Propriété 6.2.1, nous nous appuyons sur la propriété de cohérence d'un contexte pour la vérifier.

En effet, nous savons que dans un contexte donné tous les concepts non-bornés sont représentés par un CPS et tous les CPS sont compatibles. Ainsi, chacune des deux sous-configurations à lier, représentant des concepts distincts, a nécessairement dans son propre contexte un CPS représentant le concept de la deuxième sous-configuration. Si ce CPS possède un état de configuration qui peut être inclus dans l'état de configuration de la sous-configuration à lier du même concept, alors cela signifie que les deux CPS sont compatibles, comme illustré dans la Figure 6.3.

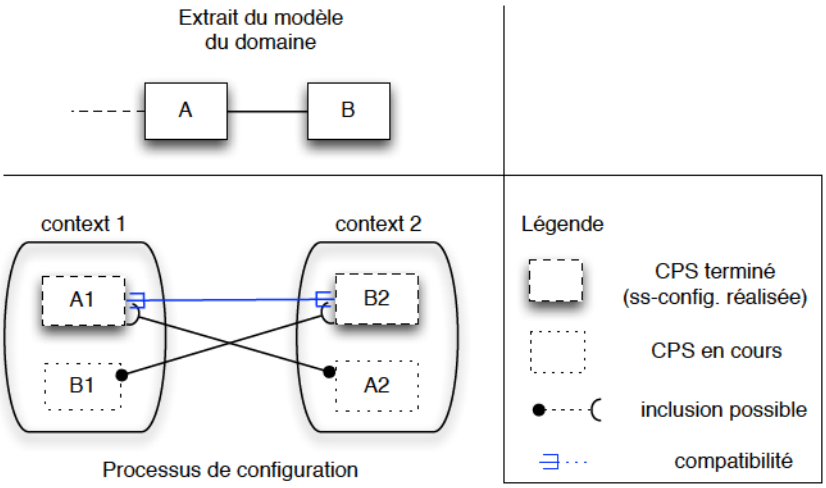


FIGURE 6.3 – Illustration du test de la compatibilité entre des CPS

Propriété 6.6.1 (Vérification de la compatibilité de CPS issues de contextes différents) : Soit $\mathcal{C}_a$ et $\mathcal{C}_b$ deux contextes locaux. Soit $\mathcal{CPS}_i^a = \langle \mathcal{D}_i, \mathcal{S}_i^a, \mathcal{SA}_i^a \rangle$ et $\mathcal{CPS}_j^b = \langle \mathcal{D}_j, \mathcal{S}_j^b, \mathcal{SA}_j^b \rangle$ deux CPS à partir desquels des configurations ont été créées et qui appartiennent respectivement à $\mathcal{C}_a$ et $\mathcal{C}_b$.

On peut alors trouver $\mathcal{CPS}_i^b = \langle \mathcal{D}_i, \mathcal{S}_i^b, \mathcal{SA}_i^b \rangle$ et $\mathcal{CPS}_j^a = \langle \mathcal{D}_j, \mathcal{S}_j^a, \mathcal{SA}_j^a \rangle$ référençant les mêmes concepts mais appartenant cette fois respectivement aux contextes $\mathcal{C}_b$ et $\mathcal{C}_a$.

S'il existe une association $\mathcal{A}_{ij}$ telle que $\mathcal{A}_{ij} \rightsquigarrow \{\mathcal{D}_i, \mathcal{D}_j\}$, alors on a :
$\mathcal{CPS}_i^a \rightleftharpoons \mathcal{CPS}_j^b \Leftrightarrow (\mathcal{S}_i^b \subseteq \mathcal{S}_i^a) \wedge (\mathcal{S}_j^a \subseteq \mathcal{S}_j^b)$.

Si les deux CPS référencés par les sous-configurations à lier sont compatibles dans des contextes distincts, cela ne signifie pas nécessairement que la liaison est possible. Il est nécessaire en effet de vérifier que les deux sous-configurations à lier ne font pas partie, chacune de leur côté, d'un assemblage qui serait incohérent.

La Figure 6.4 montre ainsi une liaison entre des sous-configurations faisant référence à des CPS qui sont compatibles mais dont l'assemblage est incohérent : les deux sous-configurations de C devraient être les mêmes pour que la liaison soit autorisée. Par ailleurs, si le maintien de contextes distincts est nécessaire afin d'assurer la flexibilité du processus de configuration, sans un raisonnement approprié il peut compromettre la cohérence du processus.

La Figure 6.5 illustre les incohérences qui peuvent résulter d'un maintien des contextes distincts. En considérant trois concepts A, B et C liés entre eux au sein d'un modèle du domaine, on considère que l'utilisateur crée deux sous-configurations A1 et B2 respectivement relatives à A et à B dans deux contextes distincts mais de façon à ce qu'elles soient compatibles pour pouvoir les lier. La seule manière de garantir la compatibilité future de la sous-configuration créée est de fusionner les CPS relatifs à C contenus dans les deux contextes : quelles que soient les actions futures réalisées, la sous-configuration sera alors forcément compatible avec les deux sous-configurations A1 et B2.

Nous considérons ainsi que l'action utilisateur de *liaison de configurations* doit, afin de garantir la flexibilité du processus de configuration, réaliser des opérations de fusion et de suppression de contextes. Il est cependant nécessaire avant d'autoriser la liaison des sous-configurations de vérifier que les contextes peuvent être fusionnés. Nous définissons la compatibilité de fusion des contextes, par le fait que deux contextes peuvent réaliser l'union de chacun de leurs CPS.

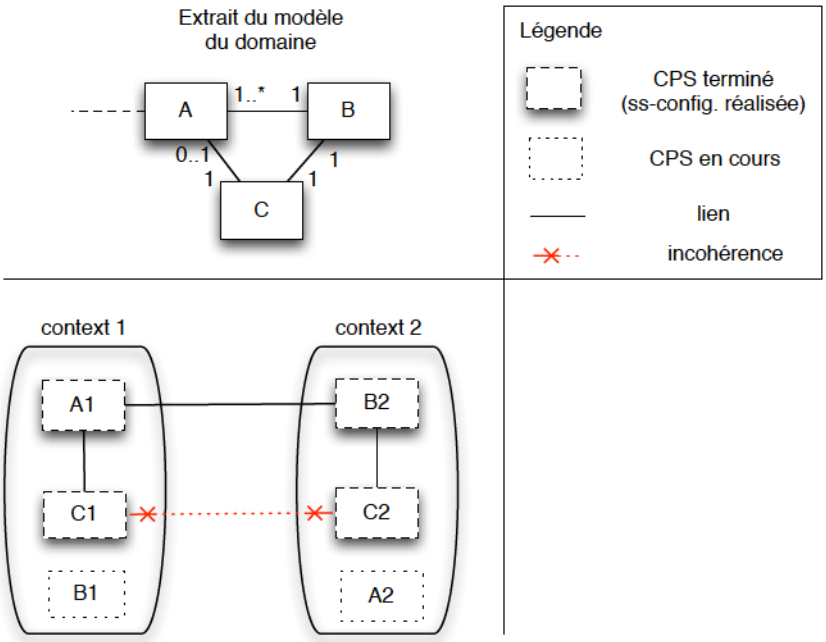


FIGURE 6.4 – Illustration d'une incohérence lors de la liaison de CPS compatibles

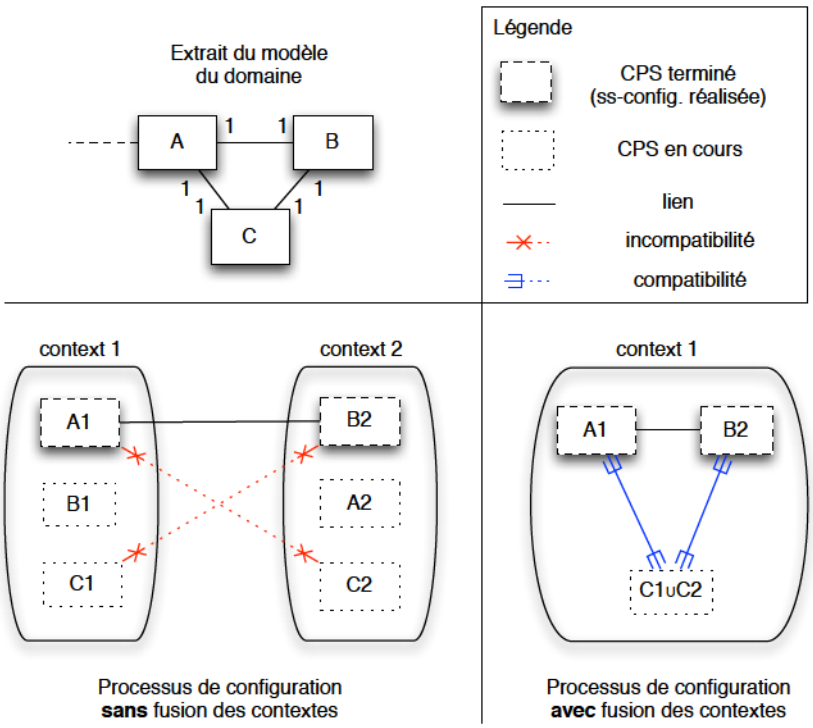


FIGURE 6.5 – Illustration du besoin de fusion des contextes

Propriété 6.6.2 (Compatibilité de fusion des contextes) :
Soit $\mathcal{C}_a = \langle SCPS_a \rangle$ et $\mathcal{CPS}_b = \langle SCPS_b \rangle$ deux contextes locaux. Nous disons que $\mathcal{C}_a$ et $\mathcal{C}_b$ sont compatibles à la fusion si et seulement si : $\forall\, \mathcal{CPS}_i^a \in SCPS_a, \exists\, \mathcal{CPS}_i^b \in SCPS_b$ référençant le même concept que $\mathcal{CPS}_i^a$ et tel que $\mathcal{CPS}_i^a \subseteq \mathcal{CPS}_i^b$ ou $\mathcal{CPS}_i^b \subseteq \mathcal{CPS}_i^a$.

Ainsi, la compatibilité de deux sous-configurations est respectée si les propriétés 6.6.1 et 6.6.2 sont vérifiées.

6.6.3.2 Fusion et suppression de contextes

Nous définissons la fusion de contextes comme une opération permettant d'obtenir à partir de deux contextes *locaux* distincts, un nouveau contexte local contenant des CPS correspondant à l'union des CPS référençant le même concept des deux contextes d'origine.

Définition 6.6.1 (Fusion de contextes) :
Soit deux contextes $\mathcal{C}_a = \langle SCPS_a \rangle$ et $\mathcal{C}_b = \langle SCPS_b \rangle$.

Nous définissons une fonction de $fusion$ prenant en argument deux contextes, telle que $fusion(\mathcal{C}_a, \mathcal{C}_b) = \mathcal{C}_c$ avec $\mathcal{C}_c = \langle SCPS_c \rangle$.

On a alors $\forall \mathcal{CPS}_i^c \in SCPS_c$ avec $\mathcal{CPS}_i^c = \langle \mathcal{D}_i, \mathcal{S}_i^c, SA_i^c, \mathcal{SC}_i^c \rangle$ $\exists \mathcal{CPS}_i^a \in SCPS_a$ et $\mathcal{CPS}_i^b \in SCPS_b$ tel que $\mathcal{CPS}_i^c = \mathcal{CPS}_i^a \cup \mathcal{CPS}_i^b$ (Propriété 6.2.3).

Ainsi, deux sous-configurations liées appartenant à des contextes *locaux* différents vont donner lieu à un troisième contexte référençant les deux sous-configurations. La question se pose alors du devenir des contextes d'origine : faut-il les conserver ou les supprimer ?

La réponse est intimement liée aux multiplicités supportées par les associations définies dans le modèle du domaine. En effet, la conservation d'un contexte après une fusion dans l'optique de créer un lien ne se justifie *que* si l'utilisateur doit pouvoir créer un autre lien vers une sous-configuration d'un concept similaire. Ainsi, la conservation d'un contexte n'est réalisée que si la multiplicité portée par l'association pour ce concept est non bornée.

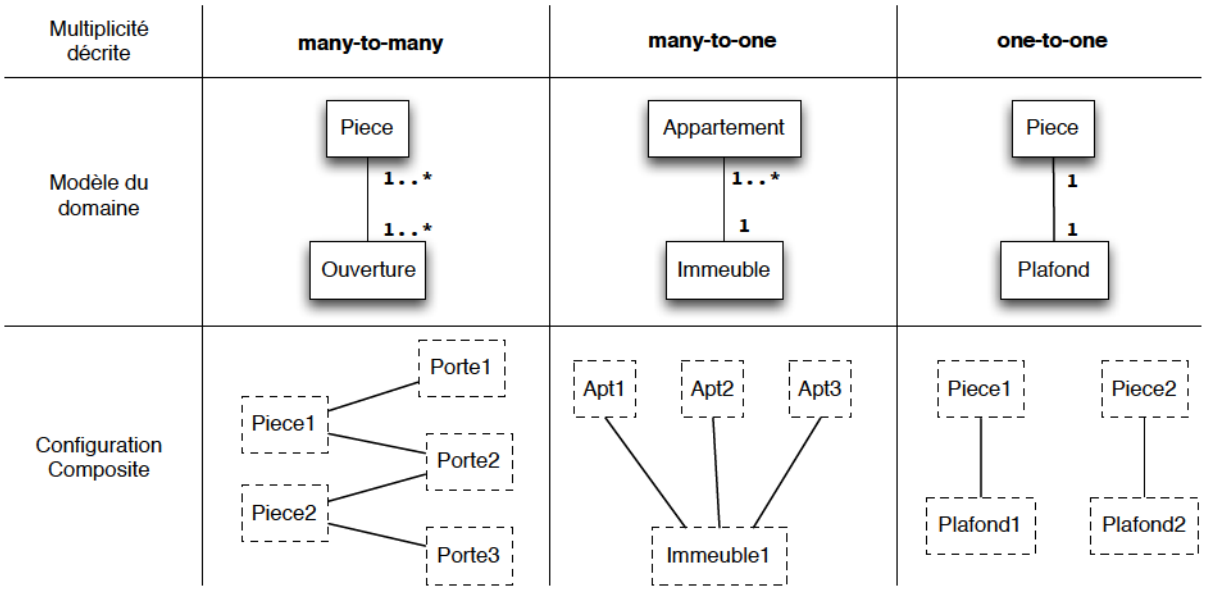


FIGURE 6.6 – Description des multiplicités rencontrées

Trois cas peuvent donc se présenter, que nous nommons par analogie avec le problème des relations dans les bases de données (voir Figure 6.6) :

1. many-to-many : les multiplicités de l'association sont non bornées pour les deux concepts : dans ce cas, les trois contextes sont conservés ;

2. many-to-one : une seule des deux multiplicités est bornée : dans ce cas, un seul des trois contextes est supprimé, celui contenant le CPS d'origine relatif au concept situé du côté borné de l'association ;
3. one-to-one : les multiplicités sont bornées pour les deux concepts : dans ce cas, seul le contexte issu de la fusion est conservé, les autres sont supprimés.

Nous maintenons ainsi la flexibilité du processus de configuration tout en garantissant sa cohérence : les contextes sont manipulés de sorte que l'utilisateur puisse continuer à réaliser des liens en cohérence avec le modèle du domaine et qu'il puisse également créer de nouvelles sous-configurations cohérentes à partir d'une sous-configuration ou d'un assemblage de sous-configurations existant.

6.6.4 Démarrage d'une sous-configuration

Une autre action utilisateur est très fortement liée à la flexibilité du processus de configuration : le *démarrage d'une nouvelle sous-configuration*.

Plusieurs cas peuvent se présenter : soit l'utilisateur souhaite créer une nouvelle sous-configuration détachée de toutes ses actions antérieures [5], soit il souhaite créer une sous-configuration dans l'optique de la lier avec une ou plusieurs déjà existantes. Dit autrement, soit il souhaite créer une sous-configuration dans un nouveau contexte, soit il souhaite la créer dans un contexte existant.

L'action de démarrage d'une nouvelle configuration prend donc en paramètre le concept pour lequel l'utilisateur souhaite créer une nouvelle sous-configuration, éventuellement le contexte dans lequel il souhaite la créer, ainsi que le concept avec lequel il souhaite lier cette configuration. En retour, l'action renvoie un CPS à partir duquel l'utilisateur pourra effectuer ses actions.

Nous nous concentrons ici sur la création d'une nouvelle sous-configuration d'un concept possédant une multiplicité non-bornée, sans quoi, la sous-configuration est directement réalisée à partir du contexte global.

6.6.4.1 Démarrage dans un nouveau contexte

Si l'utilisateur souhaite créer une nouvelle sous-configuration dans un nouveau contexte, alors les arguments de contexte et de concept à lier ne sont pas renseignés. Un nouveau contexte est alors créé en utilisant l'action système correspondante qui réalisera le contexte en clonant les CPS du contexte global référençant des concepts non-bornés. La copie des CPS du contexte global nous assure, en effet, la cohérence des choix utilisateurs.

6.6.4.2 Démarrage dans un contexte existant

En revanche, nous considérons que si l'utilisateur souhaite créer une sous-configuration dans un contexte existant, alors cela signifie qu'il a pour objectif de lier cette nouvelle sous-configuration à une autre [6] du contexte existant. Il fournit ainsi le contexte dans lequel il souhaite réaliser la nouvelle sous-configuration, ainsi que le concept auquel il souhaite la lier plus tard.

Ainsi, en fonction de l'association existante entre les deux concepts, le même raisonnement que pour la liaison des sous-configurations est effectué afin de déterminer comment créer la nouvelle sous-configuration.

5. A l'exception bien entendu des impacts dans le contexte global.
6. Cette autre sous-configuration n'est d'ailleurs pas forcément terminée à ce moment là.

Trois cas peuvent se présenter, que nous nommons également par analogie avec le problème des relations dans les bases de données :

1. one-to-one : l'association a des multiplicités bornées ;
2. many-to-one : l'association possède une unique multiplicité non-bornée du côté du concept de la nouvelle sous-configuration ;
3. X-to-many : l'association présente une multiplicité non-bornée du côté du concept de la nouvelle sous-configuration.

one-to-one Si l'association ne possède que des multiplicités bornées, la sous-configuration pourra directement être réalisée au sein du contexte existant. En effet, dans ce cas il n'est pas nécessaire de conserver un autre contexte pour permettre une future liaison : le but est de créer une sous-configuration qui pourra être immédiatement liée au sein du contexte en cours.

many-to-one Une association *many-to-one* signifie que dans le cadre de la relation il existe plusieurs exemplaires du concept à configurer, mais qu'ils sont tous liés à une unique sous-configuration. Dit autrement, l'association possède une multiplicité non-bornée, mais la multiplicité est toujours bornée sur l'autre concept de l'association.

Dans ce cas, il est nécessaire de créer un nouveau contexte afin d'autoriser de futurs liens avec le concept sur lequel porte la multiplicité bornée. Dès lors, l'action va cloner le contexte dans lequel l'utilisateur souhaite réaliser la configuration, en copiant chacun des CPS du contexte.

Finalement l'action retourne le CPS du contexte d'origine comme CPS pour réaliser la sous-configuration.

X-to-many Nous définissons par *X-to-many* le fait que nous traitons indifféremment les associations *many-to-many* et *one-to-many* : dans les deux cas il s'agit d'une relation dans laquelle un exemplaire du concept à configurer est lié de nombreuses fois à différentes sous-configurations.

Dans ces deux cas, il est nécessaire de créer un nouveau contexte afin de permettre à l'utilisateur de lier la sous-configuration à réaliser à de nombreux éléments. Cependant, l'utilisateur souhaite créer une sous-configuration qui soit *compatible* avec le contexte d'origine.

Ainsi, le contexte nouvellement créé est modifié afin que le CPS du concept à configurer soit identique à celui du contexte d'origine. Une fois cette modification effectuée, il est nécessaire de lancer l'algorithme de propagation à partir de ce CPS modifié afin de garantir la cohérence du nouveau contexte.

Finalement, l'action retourne le CPS correspondant au concept à réaliser du nouveau contexte.

6.6.5 Conclusions sur la cohérence et la flexibilité

Le processus de configuration consiste en un ensemble d'actions utilisateur permettant d'atteindre une configuration composite. Nous avons montré dans cette section comment ces actions utilisateur sont définies, aussi bien au niveau de leur précondition, qu'au niveau de leurs mécanismes d'applications afin de garantir les propriétés de cohérence et flexibilité du processus de configuration, notamment en utilisant les différentes propriétés établies dans notre formalisme ainsi que l'algorithme de propagation.

6.7 Illustration du processus de configuration

Nous illustrons les différents éléments du processus de configuration en reprenant l'exemple fil rouge que nous avons utilisé au chapitre précédent.

6.7.1 Description de l'exemple

Nous considérons dans cet exemple le modèle du domaine et les FM suivants :

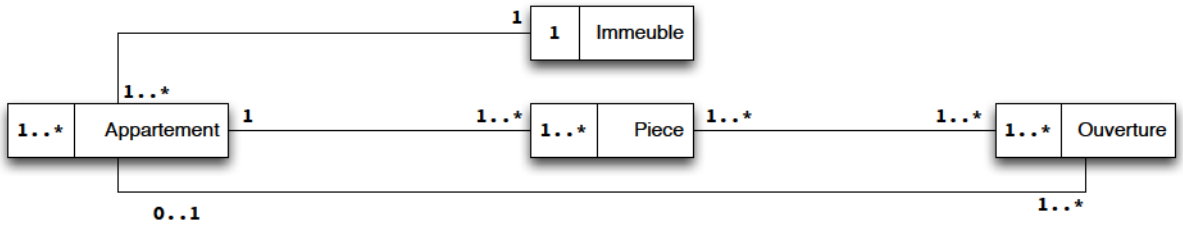


FIGURE 6.7 – Exemple du modèle du domaine pour l'exemple fil rouge

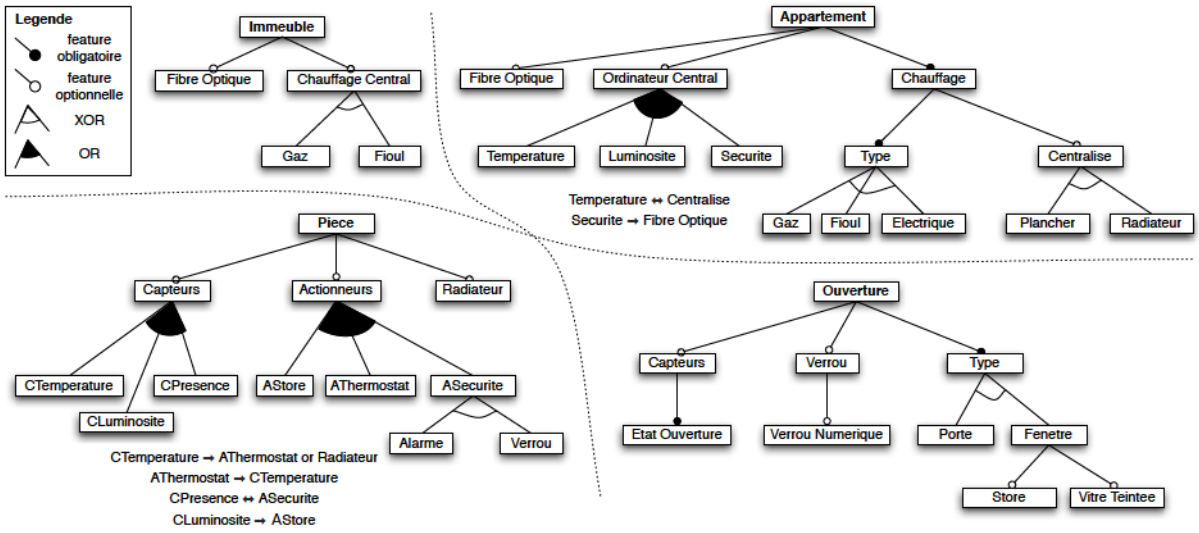


FIGURE 6.8 – Feature Models pour l'exemple fil rouge

6.7.2 Initialisation

Le processus de configuration commence avec une action d'*initialisation*. Durant cette action le contexte global, les CPS correspondant à chaque concept du modèle du domaine au sein de ce contexte, ainsi qu'une configuration composite vide sont créés. Par ailleurs, l'état des CPS est automatiquement mis à jour grâce au raisonnement dynamique effectué sur les FM pour représenter la sélection des features obligatoires. L'algorithme de propagation est également lancé à partir des CPS mis à jour au sein du contexte global afin de garantir que l'état du contexte global est cohérent à la fin de l'initialisation.

Dans notre exemple, le contexte global contient donc les informations suivantes :

Listing 6.1 – Contexte global à l'initialisation

```
CPS Immeuble
selections : [Immeuble]
exclusions : []

CPS Appartement
selections : [Appartement, Chauffage, Type]
exclusions : []

CPS Piece
selections : [Piece]
exclusions : []

CPS Ouverture
selections : [Ouverture, Type]
exclusions : []
```

Dans cet exemple, la propagation est lancée mais aucune règle ne s'applique, elle s'arrête donc immédiatement. Le GCC est désormais initialisé, l'utilisateur peut démarrer une nouvelle configuration.

6.7.3 Première sous-configuration en détails

6.7.3.1 Démarrage de la sous-configuration

L'utilisateur souhaite tout d'abord créer un nouvel *Appartement*. Il va donc réaliser une action de *démarrage d'une nouvelle sous-configuration*. Il donne comme argument le concept *Appartement* et aucune information de contexte, étant donné qu'il souhaite réaliser cette sous-configuration dans un nouveau contexte.

Une action système de création de contexte est donc lancée et un contexte local est créé que nous nommerons : cl_1. Ce contexte est un clone du contexte global, à l'exception près qu'il ne contient aucun CPS pour *Immeuble*, ce concept possédant une multiplicité de `1` dans le modèle du domaine.

Listing 6.2 – Contexte cl_1 lors de sa création

```
CPS Appartement
selections : [Appartement, Chauffage, Type]
exclusions : []

CPS Piece
selections : [Piece]
exclusions : []

CPS Ouverture
selections : [Ouverture, Type]
exclusions : []
```

6.7.3.2 Première sélection

L'utilisateur souhaite avoir une gestion centralisée de la *Temperature* dans l'*Appartement* qu'il est en train de créer. Il réalise donc une action de *Sélection* prenant pour argument la feature *Temperature* et le CPS *Appartement* du contexte cl_1. Cette action anodine va avoir un grand nombre d'impacts.

En premier lieu, le raisonnement dynamique sur la configuration du FM *Appartement* permet d'inférer automatiquement des choix à partir de la sélection de *Temperature*. Cette feature est, en effet, une feature enfant de *Ordinateur Central* : cette dernière est donc également sélectionnée

automatiquement. Par ailleurs, une contrainte du FM spécifie que *Temperature* ⇔ *Centralise*. La feature *Centralise* est donc sélectionnée aussi.

L'état du CPS *Appartement* du contexte cl_1 est donc le suivant immédiatement après l'action système de sélection de *Temperature* :

Listing 6.3 – Etat du CPS *Appartement* dans le contexte cl_1 après sélection

```
selections  : [Appartement, Chauffage, Type, Temperature, Ordinateur Central, Centralise]
exclusions  : []
```

L'action utilisateur lance ensuite l'*algorithme de propagation* à partir du CPS *Appartement* et du contexte cl_1. La propagation commence par rechercher et parcourir les associations. Le concept *Appartement* dispose de trois associations : la première avec le concept *Immeuble*, la seconde avec le concept *Pièce* et la troisième avec le concept *Ouverture*. L'algorithme itère sur les associations et récupère les concepts associés, puis elle détermine en fonction de la multiplicité quel est le CPS qui doit être utilisé.

Le concept *Immeuble* est traité en premier lieu. Comme il s'agit d'un concept à la multiplicité bornée, le CPS considéré est celui du contexte global. L'algorithme détermine ensuite si des règles s'appliquent suivant l'état du CPS *Appartement* de cl_1 : en l'occurence aucune règle ne s'applique pour le concept *Immeuble*.

La seconde association est alors considérée. Cette fois le concept *Pièce* possède une multiplicité non-bornée, le CPS considéré est directement celui du contexte cl_1. Une règle s'applique en effet sur ce concept :

Listing 6.4 – Règle pour la *Temperature* entre *Appartement* et *Piece*

```
regle appartement_piece_temperature

source  : Appartement
cible   : Piece

selections [Temperature], exclusions [] => SELECTIONNE CTemperature
```

L'action système de sélection de la feature *CTemperature* sur le CPS *Piece* du contexte cl_1 est donc appliquée. Le contexte cl_1 est désormais dans l'état suivant :

Listing 6.5 – Contexte cl_1 lors de la propagation

```
CPS Appartement
selections  : [Appartement, Chauffage, Type, Temperature, Ordinateur Central, Centralise]
exclusions  : []

CPS Piece
selections  : [Piece, CTemperature, Capteurs]
exclusions  : []

CPS Ouverture
selections  : [Ouverture, Type]
exclusions  : []
```

L'information que le CPS de *Piece* dans cl_1 a été modifié est stockée. L'association avec le concept *Ouverture* est ensuite considérée, mais comme pour l'*Immeuble* aucune règle ne s'applique.

La fonction de l'algorithme de propagation retourne donc le fait que le CPS de *Piece* dans cl_1 a été impacté par la propagation. Aucun changement n'a eu lieu dans le contexte global, l'algorithme est donc relancé à partir de ce CPS.

Le concept *Pièce* est lié au concept *Appartement* et *Ouverture*, cependant aucune règle ne s'applique sur ces concepts à partir de l'état actuel du CPS *Piece* dans cl_1. La propagation se termine donc et le contexte cl_1 est dans un état cohérent.

6.7.3.3 Deuxième sélection

L'utilisateur souhaite ensuite utiliser des radiateurs au lieu d'un parquet chauffant pour l'ensemble de l'appartement. Il sélectionne donc la feature *Radiateur* dans le CPS *Appartement* du contexte cl_1. L'algorithme de propagation est à nouveau lancé, la règle suivante sera déclenchée :

Listing 6.6 – Règle pour le *Radiateur* entre *Appartement* et *Piece*

```
regle appartement_piece_radiateur

source : Appartement
cible : Piece

selections [Radiateur], exclusions [] => SELECTIONNE Radiateur
```

L'état du contexte après la propagation est donc le suivant :

Listing 6.7 – Contexte cl_1 après la seconde sélection

```
CPS Appartement
selections : [Appartement, Chauffage, Type, Temperature, Ordinateur Central, Centralise,
Radiateur]
exclusions : []

CPS Piece
selections : [Piece, CTemperature, Capteurs, Radiateur]
exclusions : []

CPS Ouverture
selections : [Ouverture, Type]
exclusions : []
```

6.7.3.4 Troisième sélection

Notre utilisateur souhaite ensuite spécifier que l'appartement disposera d'un chauffage au *gaz*. Il sélectionne cette fois la feature *Gaz*, toujours dans le CPS *Appartement* du contexte cl_1. L'algorithme de propagation est lancé une fois encore, mais cette fois avant la propagation l'information est réifiée que les features *Fioul* et *Electrique* ont été exclues : la feature *Gaz* fait partie d'un groupe XOR avec les deux précédentes.

L'état du contexte cl_1 juste avant la propagation est donc le suivant :

Listing 6.8 – Contexte cl_1 après la troisième sélection

```
CPS Appartement
selections : [Appartement, Chauffage, Type, Temperature, Ordinateur Central, Centralise,
Radiateur, Gaz]
exclusions : [Fioul, Electrique]

CPS Piece
selections : [Piece, CTemperature, Capteurs, Radiateur]
exclusions : []

CPS Ouverture
selections : [Ouverture, Type]
exclusions : []
```

Il existe des règles de restriction entre *Immeuble* et *Appartement* spécifiant que si un *chauffage central* au *Fioul* (resp. au *Gaz*) est choisi pour l'*Immeuble*, alors tous les *Appartements* auront un chauffage au *Fioul* (resp. *Gaz*). Une règle contraposée a donc été automatiquement calculée auparavant pour spécifier que si le type de chauffage d'un *Appartement* n'est pas au *Fioul* (resp. au *Gaz*), alors l'*Immeuble* ne peut pas avoir un chauffage central au *Fioul* (resp. *Gaz*) :

Listing 6.9 – Règles pour les types de chauffage entre *Appartement* et *Immeuble*

```
regle immeuble_appartement_fioul_sel

source : Immeuble
cible : Appartement

selections [Fioul], exclusions [] => SELECTIONNE Fioul

regle immeuble_appartement_gaz

source : Immeuble
cible : Appartement

selections [Gaz], exclusions [] => SELECTIONNE Gaz

regle immeuble_appartement_fioul_sel_contra

source : Appartement
cible : Immeuble

selections [], exclusions [Fioul] => EXCLUT Fioul

regle immeuble_appartement_gaz_contra

source : Appartement
cible : Immeuble

selections [], exclusions [Gaz] => EXCLUT Gaz
```

Cette fois lors de l'étape de restriction de l'algorithme de propagation, la fonction va tester que le concept *Immeuble* a une multiplicité bornée : l'action d'exclusion de la feature *Fioul* sera donc réalisée au sein du contexte global.

Il existe par ailleurs, une règle spécifiant que si l'immeuble ne possède pas de chauffage central au *Fioul* alors aucun appartement ne peut utiliser ce type de chauffage :

Listing 6.10 – Règle pour le fioul entre *Immeuble* et *Appartement*

```
regle immeuble_appartement_fioul_desel

source : Immeuble
cible : Appartement

selections [], exclusions [Fioul] => EXCLUT Fioul
```

La propagation est donc relancée à partir du CPS d'*Immeuble* dans le contexte global et utilise cette règle pour exclure également la feature *Fioul* du CPS *Appartement* du contexte global.

Le contexte global sera donc le suivant après la propagation :

Listing 6.11 – Contexte global après la troisième action

```
CPS Immeuble
selections : [Immeuble]
exclusions : [Fioul]

CPS Appartement
selections : [Appartement, Chauffage, Type]
exclusions : [Fioul]

CPS Piece
selections : [Piece]
exclusions : []

CPS Ouverture
selections : [Ouverture, Type]
exclusions : []
```

6.7.3.5 Quatrième sélection

Enfin, notre utilisateur souhaite avoir un appartement relié à internet par la fibre optique. Il réalise donc une fois encore une action de sélection de la feature *Fibre Optique* dans le CPS *Appartement* du contexte cl_1. L'algorithme de propagation est lancé une nouvelle fois suite à cette action.

Il existe une bi-implication entre les concepts *Immeuble* et *Appartement* concernant la sélection de la feature *Fibre Optique*, celle-ci est représentée par les règles de restriction suivantes :

Listing 6.12 – Règles pour la *Fibre optique* entre *Appartement* et *Immeuble*

```
regle appartement_immeuble_fibre

source  : Appartement
cible   : Immeuble

selections [Fibre Optique], exclusions [] => SELECTIONNE Fibre Optique

regle immeuble_appartement_fibre

source  : Immeuble
cible   : Appartement

selections [Fibre Optique], exclusions [] => SELECTIONNE Fibre Optique
```

Dès lors, la propagation commence par utiliser la première règle de restriction pour sélectionner au sein du contexte global la feature *Fibre Optique* dans le CPS du concept *Immeuble*. Suite à cela, la propagation est à nouveau lancée à partir de ce CPS : la seconde règle de restriction est alors appliquée, mais cette fois au sein du contexte global : la feature *Fibre optique* est donc sélectionnée dans le CPS *Appartement* du contexte global.

A la fin de la propagation nous avons donc le contexte global suivant :

Listing 6.13 – Contexte global après la quatrième action

```
CPS Immeuble
selections  : [Immeuble, Fibre Optique]
exclusions  : [Fioul]

CPS Appartement
selections  : [Appartement, Chauffage, Type, Fibre Optique]
exclusions  : [Fioul]

CPS Piece
selections  : [Piece]
exclusions  : []

CPS Ouverture
selections  : [Ouverture, Type]
exclusions  : []
```

6.7.3.6 Exclusion

Pour finir sa configuration, l'utilisateur doit décider s'il souhaite ou non une gestion centralisée de la *luminosité* et/ou de la *sécurité* pour son appartement. Il décide de créer des actions d'exclusions pour ces deux features. L'exclusion de la feature *Luminosité* ne produit aucun impact. En revanche, comme la sélection d'un *verrou numérique* pour une *ouverture* implique la sélection de la *Sécurité* centralisée, l'exclusion de cette même feature de *Sécurité* implique l'exclusion de tout *verrou numérique* :

Listing 6.14 – Règles pour la *Sécurité* entre *Appartement* et *Ouverture*

```
regle ouverture_appartement_verrounum

source  : Ouverture
cible   : Appartement

selections [Verrou Numerique], exclusions [] => SELECTIONNE Securite

regle ouverture_appartement_verrounum_contra

source  : Appartement
cible   : Ouverture

selections [], exclusions [Securite] => EXCLUT Verrou Numerique
```

Le contexte cl_1 présente donc l'état suivant :

Listing 6.15 – Contexte cl_1 à la fin de la configuration d'*Appartement*

```
CPS Appartement
selections  : [Appartement, Chauffage, Type, Temperature, Ordinateur Central, Centralise,
Radiateur, Gaz, Fibre Optique]
exclusions  : [Luminosite, Securite, Fioul, Electrique, Plancher]

CPS Piece
selections  : [Piece, Capteurs, CTemperature, Radiateur]
exclusions  : []

CPS Ouverture
selections  : [Ouverture, Type]
exclusions  : [Verrou Numerique]
```

6.7.3.7 Validation de la sous-configuration

Le CPS du concept *Appartement* dans le contexte cl_1 a désormais un état de configuration terminé : l'utilisateur peut alors valider la sous-configuration. La configuration composite contient donc une unique sous-configuration :

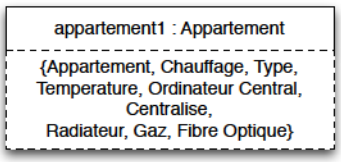


FIGURE 6.9 – Première sous-configuration de la configuration composite

6.7.4 Deuxième sous-configuration et lien

L'utilisateur souhaite ensuite créer une nouvelle sous-configuration de *Pièce* dans le contexte de la sous-configuration qu'il vient de valider, dans l'objectif de la lier avec. Il démarre donc une nouvelle sous-configuration prenant comme arguments le concept *Pièce*, le contexte cl_1 et le concept de liaison *Appartement*.

Le concept *Pièce* est lié à *Appartement* par une association "many-to-one" : l'action de démarrage d'une nouvelle sous-configuration va donc créer un nouveau contexte dans lequel le CPS correspondant au concept *Appartement* est cloné à partir du contexte d'origine. L'algorithme de propagation est automatiquement lancé pour s'assurer de la cohérence du contexte.

Nous avons donc le nouveau contexte cl_2 suivant :

Listing 6.16 – Contexte cl_2 lors de sa création

```
CPS Appartement
selections  : [Appartement, Chauffage, Type, Temperature, Ordinateur Central, Centralise,
Radiateur, Gaz, Fibre Optique]
exclusions  : [Luminosite, Securite, Fioul, Electrique, Plancher]

CPS Piece
selections  : [Piece, Capteurs, CTemperature, Radiateur]
exclusions  : []

CPS Ouverture
selections  : [Ouverture, Type]
exclusions  : [Verrou Numerique]
```

L'utilisateur souhaite créer une pièce sécurisée capable de verrouiller automatiquement les portes. Il réalise donc les actions de sélection et d'exclusion afin d'obtenir les features qui l'intéressent. Le contexte cl_2 obtenu est donc le suivant :

Listing 6.17 – Contexte cl_2 après sélections

```
CPS Appartement
selections  : [Appartement, Chauffage, Type, Temperature, Ordinateur Central, Centralise,
Radiateur, Gaz, Fibre Optique]
exclusions  : [Luminosite, Securite, Fioul, Electrique, Plancher]

CPS Piece
selections  : [Piece, Capteurs, CTemperature, Radiateur, Actionneurs, CPresence, ASecurite,
Verrou]
exclusions  : [CLuminosite, AStore, AThermostat, Alarme]

CPS Ouverture
selections  : [Ouverture, Type, Verrou, Capteurs, Etat Ouverture]
exclusions  : [Verrou Numerique]
```

Les propagations réalisées sur le CPS *Ouverture* sont liées à la *Securite* :

Listing 6.18 – Règles pour la *Sécurité* entre *Piece* et *Ouverture*

```
regle piece_ouverture_verrou

source  : Piece
cible  : Ouverture

selections [Verrou], exclusions [] => SELECTIONNE Verrou

regle piece_ouverture_etatouverture

source  : Piece
cible  : Ouverture

selections [Verrou], exclusions [] => SELECTIONNE Etat Ouverture
```

L'utilisateur valide ainsi la configuration de *Piece* réalisée, puis crée un lien avec la configuration d'*appartement* faite précédemment. L'utilisateur réalise donc une action de *liaison des configurations* prenant en paramètre les deux configurations créées. L'action va alors récupérer les CPS faisant référence aux différentes configurations pour déterminer s'ils appartiennent au même contexte. En l'occurence, lors du démarrage de la nouvelle configuration, le CPS *Appartement* a été dupliqué à partir de celui du contexte cl_1 : la référence vers la configuration a donc été également dupliquée. Il existe donc deux CPS faisant référence aux sous-configurations à lier au sein du même contexte : les deux sous-configurations sont alors nécessairement compatibles. Une action système de création d'un nouveau lien entre les sous-configurations est donc lancée.

La configuration composite est donc la suivante :

FIGURE 6.10 – Premier lien de la configuration composite

6.7.5 Gérer de nombreux contextes

Nous considérons par la suite que l'utilisateur a décidé de créer un autre appartement, plusieurs *Piece* et plusieurs *Ouverture*. La configuration composite est alors la suivante :

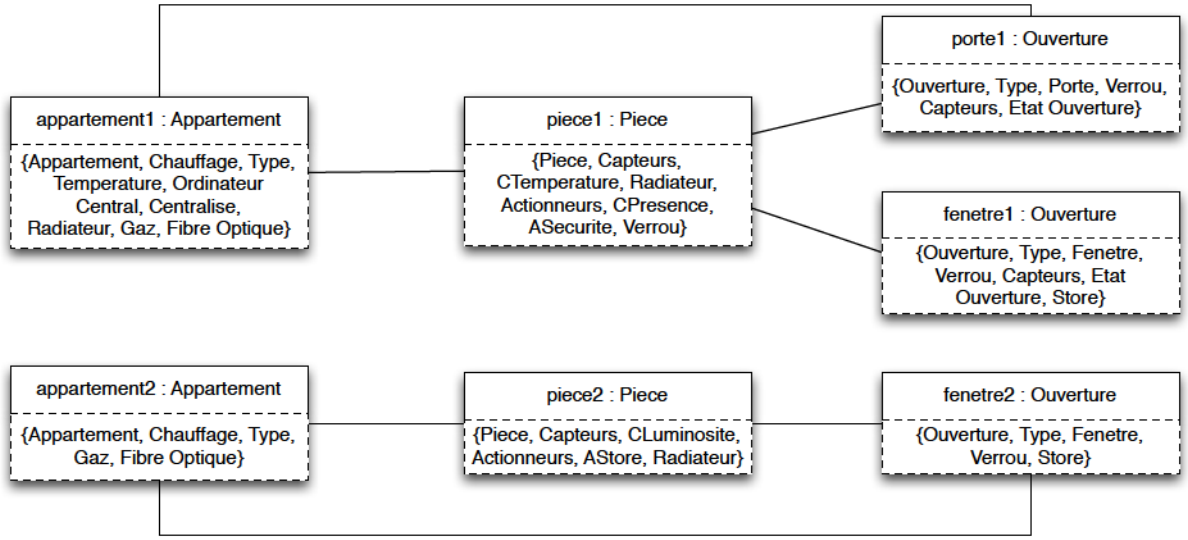


FIGURE 6.11 – Configuration composite après plusieurs configurations

Nous avons alors dix contextes locaux différents, suite aux différentes actions réalisées par l'utilisateur. Le contexte global possède le même état que celui présenté dans le Listing 6.13.

Le contexte cl_1 relatif à la configuration `appartement1` reste dans le même état que défini dans le Listing 6.15 : la cardinalité de l'association entre *Appartement* et *Ouverture* n'est pas bornée du côté de l'*Ouverture*.

L'état du contexte cl_2 est toujours le même qu'après le lien réalisé entre l'`appartement1` et la `piece1` : voir Listing 6.17.

Le contexte cl_3 permet de porter les informations relatives à la `porte1` :

Listing 6.19 – Contexte cl_3

```
CPS Appartement
selections : [Appartement, Chauffage, Type, Temperature, Ordinateur Central, Centralise,
Radiateur, Gaz, Fibre Optique]
exclusions : [Luminosite, Securite, Fioul, Electrique, Plancher]

CPS Piece
selections : [Piece]
exclusions : []

CPS Ouverture
selections : [Ouverture, Type, Porte, Verrou, Capteur, Etat Ouverture]
exclusions : [Verrou Numerique, Fenetre, Store, Vitre Teintee]
```

Ce contexte a permis la création d'une sous-configuration de *Ouverture*, comme celle-ci ne peut être liée qu'à un seul appartement le CPS de *Appartement* a été répliqué lors de la liaison des sous-configurations.

En revanche, la liaison de `porte1` avec `piece1` a créé un autre contexte dédié :

Listing 6.20 – Contexte cl_4

```
CPS Appartement
selections  : [Appartement, Chauffage, Type, Temperature, Ordinateur Central, Centralise,
Radiateur, Gaz, Fibre Optique]
exclusions  : [Luminosite, Securite, Fioul, Electrique, Plancher]

CPS Piece
selections  : [Piece, Capteurs, CTemperature, Radiateur, Actionneurs, CPresence, ASecurite,
Verrou]
exclusions  : [CLuminosite, AStore, AThermostat, Alarme]

CPS Ouverture
selections  : [Ouverture, Type, Porte, Verrou, Capteur, Etat Ouverture]
exclusions  : [Verrou Numerique, Fenetre, Store, Vitre Teintee]
```

De même il existe deux contextes pour la sous-configuration `fenetre1`. Le premier pour représenter la sous-configuration :

Listing 6.21 – Contexte cl_5

```
CPS Appartement
selections  : [Appartement, Chauffage, Type]
exclusions  : [Securite]

CPS Piece
selections  : [Piece]
exclusions  : []

CPS Ouverture
selections  : [Ouverture, Type, Fenetre, Verrou, Capteur, Etat Ouverture, Store]
exclusions  : [Verrou Numerique, Porte, Vitre Teintee]
```

Ici le CPS de l'*Appartement* n'est pas terminé car la `fenetre1` n'est liée à aucun appartement.

Et un second contexte pour la liaison entre `fenetre1` et `piece1` :

Listing 6.22 – Contexte cl_6

```
CPS Appartement
selections  : [Appartement, Chauffage, Type]
exclusions  : [Securite]

CPS Piece
selections  : [Piece, Capteurs, CTemperature, Radiateur, Actionneurs, CPresence, ASecurite,
Verrou]
exclusions  : [CLuminosite, AStore, AThermostat, Alarme]

CPS Ouverture
selections  : [Ouverture, Type, Fenetre, Verrou, Capteur, Etat Ouverture, Store]
exclusions  : [Verrou Numerique, Porte, Vitre Teintee]
```

Le contexte cl_7 présente les informations relatives à `appartement2` :

Listing 6.23 – Contexte cl_7

```
CPS Appartement
selections  : [Appartement, Chauffage, Type, Gaz, Fibre Optique]
exclusions  : [Ordinateur Central, Temperature, Luminosite, Securite, Fioul, Electrique,
Centralise, Plancher, Radiateur]

CPS Piece
selections  : [Piece]
exclusions  : []
```

```
CPS Ouverture
selections  : [Ouverture, Type]
exclusions  : []
```

Ce contexte est beaucoup plus libre étant donné que la sous-configuration `appartement2` est très peu contraignante.

Le contexte cl_8 permet de porter les informations relatives à `piece2` :

Listing 6.24 – Contexte cl_8

```
CPS Appartement
selections  : [Appartement, Chauffage, Type, Gaz, Fibre Optique]
exclusions  : [Ordinateur Central, Temperature, Luminosite, Securite, Fioul, Electrique,
Centralise, Plancher, Radiateur]

CPS Piece
selections  : [Piece, Capteurs, CLuminosite, Actionneurs, AStore, Radiateur]
exclusions  : [CTemperature, CPresence, AThermostat, ASecurite, Alarme, Verrou]

CPS Ouverture
selections  : [Ouverture, Type]
exclusions  : []
```

Enfin, le contexte cl_9 a permis de créer la sous-configuration `fenetre2` :

Listing 6.25 – Contexte cl_9

```
CPS Appartement
selections  : [Appartement, Chauffage, Type]
exclusions  : [Securite]

CPS Piece
selections  : [Piece]
exclusions  : [Verrou]

CPS Ouverture
selections  : [Ouverture, Type, Fenetre, Verrou, Store]
exclusions  : [Verrou Numerique, Porte, Vitre Teintee, Capteur, Etat Ouverture]
```

Là encore, il nous faut un autre contexte pour réaliser le lien entre `fenetre2` et `piece2` :

Listing 6.26 – Contexte cl_{10}

```
CPS Appartement
selections  : [Appartement, Chauffage, Type]
exclusions  : [Securite]

CPS Piece
selections  : [Piece, Capteurs, CLuminosite, Actionneurs, AStore, Radiateur]
exclusions  : [CTemperature, CPresence, AThermostat, ASecurite, Alarme, Verrou]

CPS Ouverture
selections  : [Ouverture, Type, Fenetre, Verrou, Store]
exclusions  : [Verrou Numerique, Porte, Vitre Teintee, Capteur, Etat Ouverture]
```

Nous avons présenté l'ensemble des contextes locaux liés à la configuration composite présentée dans la Figure 6.11 de manière linéaire par rapport aux différentes sous-configurations. Il est important cependant de se rappeler que ces sous-configurations ont pu être créées dans n'importe quel ordre : l'utilisateur peut très bien réaliser cette configuration composite en commençant par la `porte2` et terminer par la `fenetre1` tout en réalisant la même configuration et en obtenant les mêmes contextes.

L'utilisateur souhaite désormais sélectionner le chauffage central au sein de son immeuble. Il réalise donc la sélection de la feature *Chauffage central* sur le CPS *Immeuble* du contexte global.

Le contexte global possède désormais l'état suivant :

Listing 6.27 – Contexte global

```
CPS Immeuble
selections : [Immeuble, Fibre Optique, Chauffage Central, Gaz]
exclusions : [Fioul]

CPS Appartement
selections : [Appartement, Chauffage, Type, Fibre Optique, Gaz]
exclusions : [Fioul, Electrique]

CPS Piece
selections : [Piece]
exclusions : []

CPS Ouverture
selections : [Ouverture, Type]
exclusions : []
```

L'ensemble des contextes locaux est donc également impacté pour appliquer les mêmes modifications que celles réalisées sur le CPS *Appartement* du contexte global. Par exemple les contextes locaux cl_5, cl_6, cl_9 et cl_{10} auront un CPS possédant l'état suivant pour *Appartement* :

Listing 6.28 – Etat du CPS *Appartement* dans les contextes cl_5, cl_6, cl_9 et cl_{10}

```
CPS Appartement
selections : [Appartement, Chauffage, Type, Fibre Optique, Gaz]
exclusions : [Fioul, Electrique, Securite]
```

6.7.6 Annulation d'action

L'utilisateur a la possibilité à n'importe quel moment d'annuler une action effectuée précédemment. Il souhaite par exemple annuler l'action de sélection de la feature *Gaz* lors de la réalisation de la sous-configuration `appartement1` dans le contexte cl_1 (voir sous-sous-section 6.7.3.4).

L'annulation de cette action spécifique va avoir un nombre important d'impacts. Cette action a donné lieu à plusieurs actions système de sélection et d'exclusions de features dues à la propagation qui s'en est suivie.

Par ailleurs l'utilisateur a également effectué de nombreuses actions de configurations après cette action. L'algorithme d'annulation (voir sous-section 6.5.3) va commencer par annuler une par une, les actions utilisateur réalisées, annulant également les actions système associées. Le système est donc remis dans l'état dans lequel il était avant la réalisation de cette action de sélection de la feature *Gaz*.

Cette action utilisateur et les actions système associées sont supprimées. Puis l'algorithme vérifie pour chacune des actions utilisateur suivantes si les préconditions sont toujours vérifiées et si elles peuvent être à nouveau réalisées.

L'action utilisateur suivante consistait à sélectionner la feature *Fibre optique* pour la création de la sous-configuration de `appartement1` : la précondition est toujours respectée, cette action est réalisée à nouveau. De même, les deux actions suivantes consistant à exclure les features *Luminosité* et *Sécurité* peuvent également être appliquées.

Finalement, l'utilisateur avait réalisé une action de validation de la configuration : il manque cependant désormais un choix sur le *type de chauffage* dans le CPS d'*Appartement* du contexte cl_1. L'état de configuration du CPS n'étant pas complet, les préconditions de l'action de validation de la configuration ne sont pas réalisées pour le CPS *Appartement* du contexte cl_1 : la sous-configuration n'est donc pas réalisée.

Suite à cette action l'utilisateur avait créé une configuration de *Piece* : cette sous-configuration peut être réalisée sans aucun problème. En revanche, après avoir validé la sous-

configuration `piece1`, l'utilisateur avait réalisé la liaison des sous-configurations `appartement1` et `piece1`. Désormais, la configuration composite ne contient que la sous-configuration `piece1` : une des préconditions n'est donc pas respectée pour appliquer l'action de liaison des configurations. Suite à cela, le reste des actions utilisateur peut se dérouler normalement jusqu'à la dernière action utilisateur.

Désormais, l'utilisateur a la possibilité de choisir pour le chauffage centralisé de son immeuble entre *Gaz* et *Fioul* : n'ayant pas sélectionné le *Gaz* pour le premier appartement, il n'a pas été forcé de faire ce choix par le jeu des propagations.

6.7.7 Finalisation

L'utilisateur choisit une nouvelle fois la feature *Gaz* pour le chauffage centralisé de l'*Immeuble* : l'état du système revient donc à celui envisagé avant l'annulation de l'action.

L'utilisateur a désormais la possibilité de valider sa sous-configuration d'*Immeuble* et de créer des liens avec les sous-configurations d'*Appartement*. Il réalise également le lien manquant entre l'`appartement2` et la `fenetre2`. La configuration composite obtenue est alors la suivante :

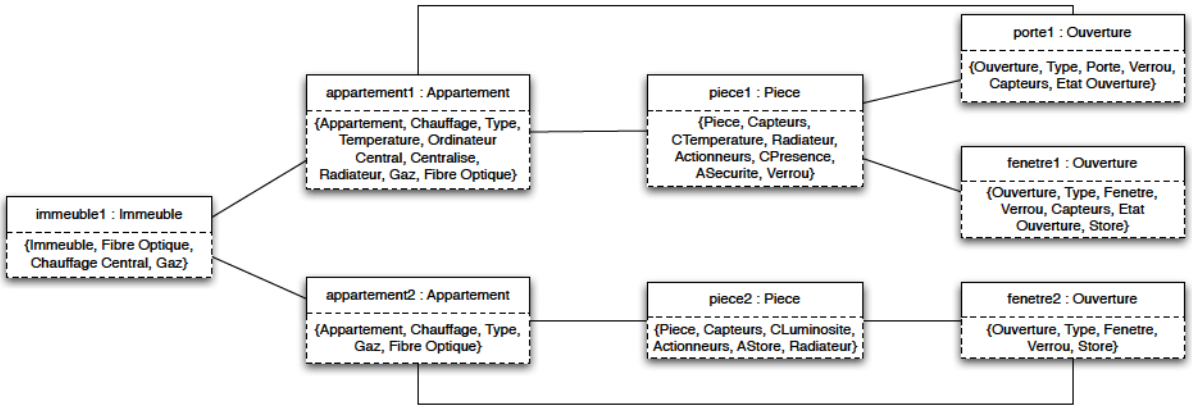


FIGURE 6.12 – Configuration composite finale

Cette configuration composite est une instance valide du modèle du domaine représenté dans la Figure 6.7, les sous-configurations sont toutes des configurations complètes et valides des FM représentés dans la Figure 6.8 et les différentes règles de restriction données à travers notre illustration ont été respectées.

Elle est donc valide et terminée selon la Propriété 5.5.5. Cette configuration composite pourra ensuite être exploitée lors d'un processus de transformation visant à l'obtention d'un produit (par exemple, à travers un outil de transformation de modèles).

6.8 Algorithme de vérification de la réalisabilité

La réalisabilité (Propriété 5.6.1) est une propriété essentielle de notre LPL que nous voulons pouvoir garantir en permanence. Nous avons discuté dans le chapitre précédent (voir sous-section 5.6.4) de la possibilité de définir un algorithme naïf pour la calculer. Cependant celui-ci aurait une complexité très grande de l'ordre de $\frac{n^k}{k^{k-2}}$ où n représente le nombre total de sous-configurations possibles dans la LPL et k représente le nombre de concepts.

Nous proposons dans cette section un algorithme incrémental qui repose sur la topologie du modèle du domaine et la connaissance des composantes biconnexes [7] du graphe ainsi représenté afin de vérifier cette propriété.

6.8.1 Principe

La vérification de la réalisabilité consiste à chercher des configurations composites minimales valides pour toutes les sous-configurations possibles du modèle du domaine. Nous considérons que si l'on connaît déjà des configurations composites minimales valides, à la suite de tests de réalisabilité déjà effectués, il est alors possible de réutiliser cette connaissance pour tester une nouvelle sous-configuration sans effectuer à nouveau un parcours exhaustif de toutes les combinaisons possibles.

En particulier, si l'on considère le modèle du domaine comme un graphe non-orienté dans lequel les concepts représentent les nœuds et les associations représentent les arêtes, alors les composantes biconnexes de ce graphe ont des propriétés de stabilité intéressantes. En effet, une composante biconnexe traduit une composante telle que si l'on enlève un unique lien, celle-ci reste connexe [Lacomme 03]. Ainsi, si une composante est biconnexe au sein d'une configuration composite minimale valide [8], cela signifie que toutes les sous-configurations de la composante sont compatibles entre elles, *quelles que soient* les sous-configurations des autres composantes de la configuration composite.

Si une sous-configuration à tester appartient à une composante biconnexe et est valide avec des sous-configurations de la composante qui ont déjà été testées auparavant, alors cette sous-configuration pourra nécessairement être incluse dans une configuration composite minimale valide. L'utilisation des composantes biconnexes dans notre algorithme de réalisabilité et la réutilisation des connaissances agrégées peuvent donc apporter un gain notable sur la complexité.

6.8.2 Illustration

La Figure 6.13 illustre l'algorithme de réalisabilité que nous proposons. Nous représentons dans cette figure chaque concept par un grand cercle contenant un ensemble de sous-configurations représentées par les petits cercles. Nous pouvons visualiser immédiatement un graphe dans cette figure : les grands cercles représentant les concepts en sont les nœuds, et les traits entre ces cercles décrivant les associations entre les concepts sont les arêtes du graphe. Ainsi, ce graphe comporte deux composantes biconnexes : la composante regroupant *Immeuble* et *Appartement*, ainsi que la composante regroupant *Appartement*, *Piece* et *Ouverture*. Le seul point d'articulation est donc le nœud *Appartement*.

Les cercles verts dans la 1ère vignette de la figure représentent l'ensemble des sous-configurations qui ont déjà été validées (c'est-à-dire pour lesquelles il existe une configuration composite minimale valide). Les cercles blancs restent donc à valider : nous considérons ici que nous avons encore à valider une sous-configuration d'*Immeuble*, deux sous-configurations d'*Appartement*, trois sous-configurations de *Piece* et deux sous-configurations d'*Ouverture*.

L'algorithme utilise pour commencer une heuristique basée sur le nombre de sous-configuration à tester dans un concept donné, ou le hasard si le nombre est identique dans les différents

7. On trouve dans la littérature indifféremment le terme biconnexe ou 2-connexe (parfois écrit bi-connexe) pour un graphe k-connexe avec k=2. Nous utiliserons invariablement le terme *biconnexe* pour cette propriété.

8. Une configuration composite minimale valide reflète forcément la topologie du modèle du domaine et donc *a fortiori* le graphe qu'il représente mais cette fois les sous-configurations représentent les nœuds et les liens représentent les arêtes.

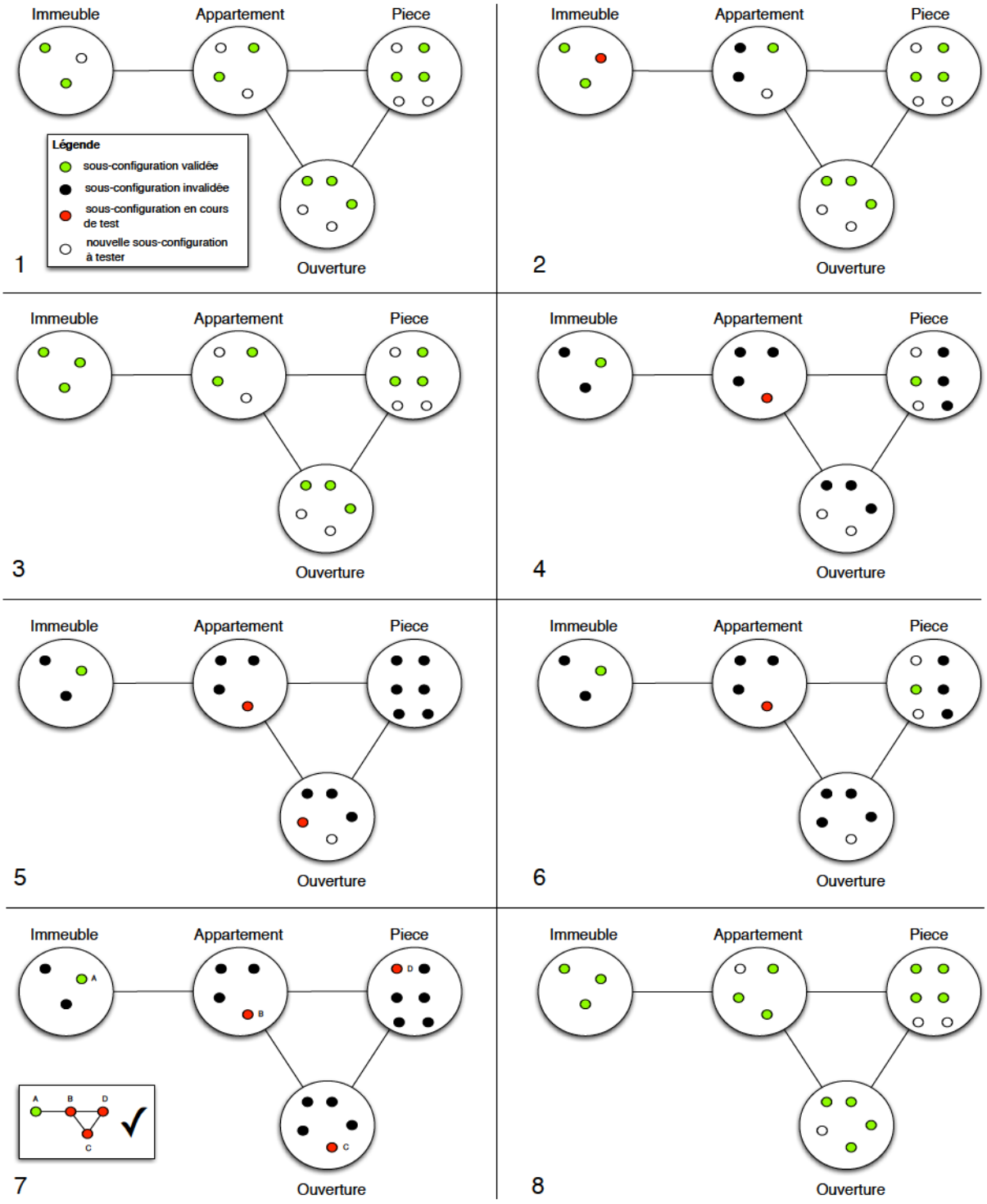


FIGURE 6.13 – Illustration de l'algorithme de réalisabilité

concepts. L'algorithme commence donc ici avec une sous-configuration d'**Immeuble** colorée en rouge (voir 2ème vignette de la Figure 6.13), étant donné qu'il s'agit du concept avec le moins de sous-configuration à tester. Nous proposons d'utiliser un algorithme de restriction inspiré de l'algorithme de propagation (voir section 6.4) afin d'éliminer automatiquement les sous-configurations incompatibles dans les concepts associés, selon les règles de restrictions : ces sous-configurations sont représentées par un cercle plein noir dans la figure. Ainsi, il ne reste dans la vignette 2 que deux sous-configurations possibles pour *Appartement*. Or, on constate qu'une

des deux sous-configurations compatibles a déjà été validée lors d'un précédent test. Il existe donc une solution dans la composante biconnexe constituée d'*Appartement*, *Piece* et *Ouverture* qui inclut cette sous-configuration et qui est valide. La sous-configuration d'*Immeuble* peut donc être incluse dans une configuration composite minimale valide : l'algorithme peut s'arrêter là pour cette sous-configuration et la considérer comme valide, comme nous le représentons en vignette 3.

Une autre sous-configuration est ensuite choisie au hasard entre les concepts *Appartement* et *Ouverture* s'agissant des concepts pour lesquels il reste le moins de sous-configurations à tester. Dans notre exemple la sous-configuration choisie est une sous-configuration du concept *Appartement*, colorée en rouge, à partir de laquelle la restriction est à nouveau appliquée (voir vignette 4 de la Figure 6.13). Cette fois, la restriction est appliquée sur tous les concepts, étant donné qu'*Appartement* est associé à tous les autres concepts.

Il existe des solutions dans tous les concepts : la sous-configuration n'est donc pas immédiatement invalidée. Par ailleurs, comme toutes les sous-configurations de *Immeuble* ont été testées et que la composante biconnexe ne contient que *Immeuble* et *Appartement*, on peut désormais ne s'intéresser qu'à la composante biconnexe incluant *Immeuble*, *Piece* et *Ouverture*. Pour continuer, l'algorithme doit sélectionner une autre sous-configuration soit dans *Piece* soit dans *Ouverture* : le nombre de sous-configurations non testé est le même, la sélection se fait donc au hasard.

Une sous-configuration d'*Ouverture* est sélectionnée (voir figure 5 de la Figure 6.13). On constate cette fois qu'il n'existe plus aucune sous-configuration de *Piece* compatible après application des restrictions. L'algorithme retourne donc en arrière et annule la restriction effectuée avant de déclarer la sous-configuration sélectionnée dans *Ouverture* comme incompatible, en vignette 6.

L'étape de la vignette 5 est appliquée à nouveau mais cette fois une configuration reste compatible pour le concept *Piece* : une configuration composite minimale valide a donc été trouvée (voir vignette 7 de la Figure 6.13).

Les sous-configurations de cette configuration composite sont donc validées et l'algorithme peut continuer à partir d'une autre sous-configuration à tester.

Nous proposons donc un algorithme qui va permettre de limiter le test des sous-configurations, grâce à la connaissance des composantes biconnexes du modèle du domaine.

6.8.3 Description de l'algorithme

Notre algorithme est un processus en deux étapes. La première étape n'est réalisée qu'une unique fois quand le modèle du domaine est défini et va permettre de déterminer ses points d'articulation, soit les nœuds communs à plusieurs composantes biconnexes [Lacomme 03].

Nous proposons pour cela d'utiliser l'algorithme de Hopcroft et Tarjan qui calcule les points d'articulation avec une complexité de l'ordre $0(n + m)$ où n représente le nombre de concepts du modèle (soit le nombre de nœuds) et m représente le nombre d'associations (le nombre d'arêtes) [Hopcroft 73].

Nous considérons par la suite que nous disposons d'une structure de données associant pour chaque concept du modèle du domaine le (ou les s'il s'agit d'un point d'articulation) graphe(s) biconnexe(s) contenant ce concept.

La deuxième étape est un algorithme complexe de parcours utilisant des heuristiques telles que le minimum de sous-configurations à tester pour un concept donné, et les composantes biconnexes calculées précédemment. Nous décrivons cet algorithme dans la suite en le décomposant en plusieurs fonctions distinctes, afin de faciliter sa compréhension.

Algorithme principal de test de la réalisabilité Nous présentons en premier lieu l'algorithme principal de test de la réalisabilité (voir Algorithme 8). Celui-ci prend en paramètre une liste de sous-configurations à tester et retourne une liste de sous-configurations invalidées. Si la liste retournée est vide, cela signifie que toutes les sous-configurations ont été validées. Par ailleurs, si la liste donnée en argument ne contient qu'un sous-ensemble des sous-configurations présentes dans le système, alors l'algorithme considère que les autres ont déjà été validées par le passé.

Cet algorithme se veut incrémental dans le sens où il réutilise les tests déjà effectués précédemment soit au cours de l'algorithme, soit lors d'un précédent lancement de l'algorithme (il est alors utilisé lors de l'évolution de la LPL).

Algorithme 8 testRealisability(inout configToTest : List<SubConfiguration>) : List<SubConfiguration>

```
1 : result := new List()
2 : mapConfigPerConcept := initMapConcept(configToTest)
3 : repeat
4 :     concept := getConceptMinimumConfig(mapConfigPerConcept)
5 :     listConfig := mapConfigPerConcept.get(concept)
6 :     config := listConfig.removeFirst()
7 :     mapConfigPerConcept.delete(config)
8 :     listeComposantes := getComposantesBiconnexe(concept)
9 :     parcours := new Map()
10 :    if testConfigAllComponent(config, concept, listeComposantes, result, parcours) then
11 :        for all configValid ∈ parcours.values() do
12 :            if configValid ∈ configToTest then
13 :                configToTest.remove(configValid)
14 :                mapConfigPerConcept.remove(configValid)
15 :    else
16 :        configToTest.remove(config)
17 :        mapConfigPerConcept.remove(configValid)
18 :        result.add(config)
19 : until configToTest.isEmpty()
20 : return result
```

L'algorithme commence par créer une liste de résultat vide (`result`) et une structure pour organiser les sous-configurations à tester par concept (`mapConfigPerConcept`) grâce à une fonction (`initMapConcept`) itérant sur les sous-configurations et qui récupère l'information du concept représenté. L'algorithme entre ensuite dans une boucle qui se finit lorsque la liste des sous-configurations à tester est vide (voir ligne 19).

La boucle principale de l'algorithme commence par chercher le concept qui doit être considéré en premier : on se base pour cela sur le concept pour lequel il y a le moins de sous-configurations à tester, via la fonction `getConceptMinimumConfig` qui va inférer cette information directement du dictionnaire organisant les sous-configurations à tester par concept. A partir de ce concept, la liste des sous-configurations à tester du concept est obtenue et la première sous-configuration de cette liste est considérée. La liste des composantes biconnexes associées au concept est ensuite obtenue à partir de la structure de données calculée avant le lancement de l'algorithme.

L'algorithme initialise ensuite un dictionnaire correspondant au parcours et lance une fonction de test de la réalisabilité de la sous-configuration considérée pour toutes ses composantes

(voir ligne 11). Nous décrivons cette fonction de test dans la suite. Si le test renvoie une valeur $vraie$, alors l'algorithme boucle sur toutes les sous-configurations parcourues durant le test, car elles constituent une configuration composite minimale valide : elles doivent donc toute être validées. Chaque sous-configuration ainsi considérée qui appartient à la liste des configurations à tester, est enlevée de cette même liste (voir ligne 13), ce qui suffit à la considérer comme valide. On notera que le test d'une seule sous-configuration peut ainsi permettre de valider plusieurs sous-configurations.

En revanche, si le test a renvoyé une valeur $fausse$ alors la sous-configuration qui était considérée n'est pas valide : elle est alors supprimée également des sous-configurations à tester, mais elle est ajoutée à la liste des sous-configurations invalidées (voir ligne 18). L'algorithme retourne finalement la liste des sous-configurations invalidées, une fois que toutes les sous-configurations à tester l'ont été.

Algorithme 9 testConfigAllComponent(config :SubConfiguration, concept :Concept, listComponents :List<List<Concept», invalidConfig :List<SubConfig>, parcours :Map<Concept,SubConfiguration>) :boolean

```
1 : gcc := initGCC()
2 : for all composante ∈ listComponents do
3 :     parcoursOneComponent := new Map()
4 :     if not testConfig(config, concept, composante, result, parcoursOneComponent, gcc) then
5 :         return false
6 :     parcours.addAll(parcoursOneComponent)
7 : return true
```

Algorithme de test d'une sous-configuration pour toutes les composantes Cet algorithme décrit dans l'Algorithme 9 est utilisé pour tester la réalisabilité d'une sous-configuration pour toutes ses composantes biconnexes. La fonction prend ainsi les arguments suivants : la sous-configuration à tester (`config`), le concept auquel elle correspond (`concept`), la liste des composantes biconnexes (`listComponents`), la liste des sous-configurations qui ont été invalidées (`invalidConfig`), le dictionnaire contenant l'ensemble des informations de parcours (`parcours`) et un GCC permettant de gérer un contexte et les CPS associés durant le parcours (`gcc`).

L'algorithme commence par initialiser un GCC qui va permettre de gérer les CPS et concepts associés durant le parcours. Puis, l'ensemble des composantes biconnexes sont parcourues et pour chaque composante, la réalisabilité de la sous-configuration est testée grâce à une fonction que nous décrivons dans la suite. Si la sous-configuration n'est pas valide pour une composante biconnexe, alors la fonction retourne immédiatement une valeur fausse. Sinon, l'ensemble des informations de parcours sont stockées et la boucle se poursuit.

Si la sous-configuration est réalisable pour chacune des composantes biconnexes (si la boucle se termine) alors la fonction retourne une valeur vraie.

Algorithme de test d'une sous-configuration pour une composante Nous définissons une fonction permettant de tester la validité d'une sous-configuration pour une composante donnée. Cette fonction est décrite dans l'Algorithme 10.

Il s'agit d'un algorithme récursif qui exploite une fonction de restriction issue de la propagation présentée dans la section 6.4. Le principe de la fonction consiste à sélectionner la configuration, effectuer la restriction, vérifier si l'ensemble des concepts à parcourir dans la

Algorithme 10 testConfig(config :SubConfiguration, concept :Concept, composante :List<Concept>, invalidConfig :List<SubConfiguration>, parcours :Map<Concept,SubConfiguration>, gcc :GCC) :boolean

```
1 : parcours.put(concept, config)
2 :
3 : if parcoursRestant(parcours, composante) = ∅ then
4 :     return true
5 :
6 : globalContext := gcc.getGlobalContext()
7 : cpsConcept := globalContext.getCPSForConcept(concept)
8 : selectionnerConfiguration(config, cpsConcept)
9 :
10 : actionsRestriction := localRestriction(cpsConcept, globalContext)
11 :
12 : potentialConfig := createMap()
13 : if not checkRemainings(composante, parcours, globalContext, invalidConfig, potentialConfig) then
14 :     undo(actionsRestriction)
15 :     parcours := parcours \ (concept, config)
16 :     return false
17 : nextConcept := getConceptMinimumConfig(potentialConfig)
18 : listConfigs := potentialConfig.get(nextConcept)
19 : repeat
20 :     configToTest := listConfigs.removeFirst()
21 :     potentialConfig.remove(configToTest)
22 :     if testConfig(configToTest, nextConcept, composante, invalidConfig, parcours, gcc) then
23 :         return true
24 : until listConfigs = ∅
25 :
26 : undo(actionsRestriction)
27 : parcours := parcours \ (concept, config)
28 : return false
```

composante ont toujours des sous-configurations compatibles, puis sélectionner le concept ayant le moins de sous-configurations compatibles pour les explorer afin de découvrir une combinaison de sous-configurations valide, soit une configuration composite minimale valide.

La fonction prend les arguments suivants en paramètre : la sous-configuration à tester (`config`), le concept associé à cette sous-configuration (`concept`), la composante sur laquelle on veut tester cette sous-configuration (`composante`), la liste des sous-configurations qui ont été invalidées (`invalidConfig`), un dictionnaire regroupant les informations des concepts et des sous-configurations parcourues (`parcours`) et enfin un GCC permettant d'exploiter un contexte et ses CPS (`gcc`).

Le début de la fonction consiste à stocker les informations de parcours : nous sommes en train d'explorer une sous-configuration spécifique pour un concept donné. Nous pouvons vérifier à partir des informations du parcours si l'ensemble de la composante a déjà été exploré, dans ce cas on considère que la sous-configuration est valide : cela ne peut se produire que si l'algorithme a trouvé uniquement des sous-configurations compatibles pour l'ensemble des

concepts de la composante. Il s'agit donc d'une condition d'arrêt de notre algorithme récursif (voir ligne 4).

Nous récupérons ensuite le contexte global du GCC ainsi que le CPS correspondant au concept de la sous-configuration à tester dans le contexte global. Nous n'avons, en effet, besoin dans cet algorithme que du contexte global pour effectuer nos propagations : nous ne souhaitons obtenir qu'une seule combinaison de sous-configurations. Nous réalisons ensuite les sélections nécessaires à la réalisation de la sous-configuration à tester dans le CPS du contexte global (voir ligne 8).

A cette étape du processus, nous avons un contexte global dans lequel une sous-configuration est sélectionnée pour un concept donné. Nous lançons donc une fonction de restriction locale, décrite dans la suite, afin d'éliminer toutes les sous-configurations qui seraient incompatibles pour les concepts associés (voir ligne 10).

Nous appelons ensuite une méthode, que nous décrivons également plus loin, qui va vérifier que pour chacun des concepts restant à parcourir de la composante, il existe bien au moins une sous-configuration restante qui n'ait pas été invalidée. Si ce n'est pas le cas, alors la sous-configuration n'est pas valide : nous annulons les restrictions grâce à une opération d'annulation comme nous l'avons défini dans la sous-section 6.5.3 (`undo`) et enlevons les informations de la sous-configuration du parcours (voir ligne 15).

La suite de la fonction consiste à sélectionner le concept pour lequel il existe le moins de sous-configurations restantes (voir ligne 17). En effet, cela permet de limiter le nombre de tests à effectuer : si ces sous-configurations ne sont pas valides, alors la sous-configuration à tester ne le sera pas non plus. Une fois ce concept sélectionné, nous itérons sur chacune de ces sous-configurations restantes afin de les tester indépendamment lors d'un appel récursif.

Enfin, si aucune sous-configuration du concept ne s'est avérée valide, alors la sous-configuration d'origine n'est pas valide non plus : nous annulons donc les opérations de restriction grâce au `undo` et enlevons les informations à propos de la sous-configuration dans le parcours (voir ligne 27).

Algorithme 11 localRestriction(cps :CPS, cont :Context) :List<Action>

```
 1 : result := new List()
 2 : conceptSrc := cps.getConcept()
 3 : assoSrc := conceptSrc.getAssociations()
 4 : for all asso ∈ assoSrc do
 5 :     conceptTarget := assoSrc.getOtherEnd(conceptSrc)
 6 :     cpsTarget := cont.getCPSofConcept(conceptTarget)
 7 :     setOfActionsToDo := getActions(asso, cps)
 8 :     for all action in setOfActionsToDo do
 9 :         if cpsTarget.getActionsDone() ∩ action = ∅ then
10 :             result.add(action)
11 :             action.apply(cpsTarget)
12 : return result
```

Fonction de restriction locale La fonction de restriction locale présentée dans l'Algorithme 11 est très similaire à la fonction de restriction utilisée dans l'algorithme de propagation (voir section 6.4). Les trois premières lignes de l'algorithme permettent respectivement de créer une liste de résultat vide, de récupérer le concept lié au CPS passé en paramètre et de récupérer l'ensemble des associations dans lequel le concept est impliqué.

La suite de l'algorithme se fait en considérant chacune des associations ainsi récupérée (ligne 4). La ligne 5 de l'algorithme permet de récupérer le deuxième concept de l'association considérée. Nous récupérons ensuite à la ligne 7 la liste des actions à effectuer sur le CPS, grâce à une fonction présentées précédemment (voir Algorithme 3).

Puis nous itérons sur chacune des actions en vérifiant à la ligne 9 si l'action a déjà été réalisée ou non au sein du CPS. Cela nous permet dans la ligne suivante d'ajouter le CPS à la liste des CPS impactés si l'action n'a pas encore été réalisée. Nous appliquons alors l'action sur le CPS considéré.

Le but de la fonction est de propager aux concepts liés les actions à réaliser étant données les règles de restrictions et l'état du CPS donné en entrée. La différence entre cette fonction et celle définie dans la propagation (voir Algorithme 2) réside en deux points :

1. cette fonction ne considère pas les multiplicités des contextes étant donné que l'algorithme de réalisabilité ne considère qu'un unique contexte ;
2. la fonction retourne une liste d'actions système afin de pouvoir les annuler dans la fonction de test principale décrite au dessus.

Algorithme 12 checkRemainings(composante : List<Concept>, parcours : Map<Concept,SubConfiguration>, globalContext : Context, invalidConfig : List<SubConfiguration>, configPotentielles : Map<Concept, SubConfiguration>) : boolean

```
1 : listConcept := finDeParcours(parcours, composante)
2 : for all concept ∈ listConcept do
3 :     cps := globalContext.getCPSForConcept(concept)
4 :     listConfig := getConfigurationsFromCPS(cps)
5 :     listConfig := listConfig \ invalidConfig
6 :     if listConfig = ∅ then
7 :         return false
8 :     else
9 :         configPotentielles.put(concept, listConfig)
10 : return true
```

Test de présence de sous-configurations potentielles La fonction présentée dans l'Algorithme 12 permet de tester si des concepts non parcourus d'une composante donnée disposent encore d'au moins une sous-configuration potentiellement valide dans un contexte global donné.

La fonction prend ainsi en paramètre la composante explorée (`composante`), la liste des éléments déjà parcourus (`parcours`), le contexte global utilisé (`globalContext`), la liste des sous-configurations invalidées, ainsi qu'un dictionnaire à remplir qui contiendra *in fine* les informations sur les configurations potentielles encore valides pour les différents concepts (`configPotentielles`).

La fonction commence par récupérer la liste des concepts de la composante qui n'ont pas encore été parcourus. Puis, pour chacun des concepts, elle récupère son CPS à partir du contexte global et calcule la liste des configurations encore disponibles (voir ligne 4). La liste ainsi calculée est réduite pour supprimer les sous-configurations déjà invalidées. Si une des liste est vide, alors il existe un concept pour lequel aucune configuration compatible n'est disponible : la configuration qui était alors en cours de test est nécessairement invalide et on peut quitter immédiatement la fonction. Sinon la fonction enregistre les listes obtenues au sein d'un dictionnaire qui sera exploité dans la fonction de test principale (voir Algorithme 10).

6.8.4 Propriétés

Nous évaluons ici deux propriétés de notre algorithme : sa capacité à terminer, étant donné qu'il exploite un mécanisme de parcours récursif, et sa complexité, étant donné qu'il a été imaginé pour améliorer les performances de l'algorithme naïf dont la complexité a été évaluée à l'ordre O($\frac{n^k}{k^{k-2}}$) avec n le nombre total de sous-configurations à évaluer et k le nombre de concepts dans le modèle du domaine (voir sous-section 5.6.4).

6.8.4.1 Terminaison

Nous devons uniquement évaluer ici la terminaison de la fonction *testConfig* décrite dans l'Algorithme 10. En effet, il s'agit de la fonction qui est au cœur notre algorithme de test de la réalisabilité et c'est la seule fonction récursive que nous employons. Par ailleurs, les listes itérées au sein des boucles dans les autres fonctions ne sont pas modifiées par l'algorithme, ce qui garantit leur terminaison.

Cette fonction possède une première condition d'arrêt qui consiste à vérifier si l'ensemble des concepts de la composante a été parcouru ou non. Nous garantissons ainsi l'arrêt de la fonction au-delà d'un certain nombre de concepts visités.

Par ailleurs, la suite de l'exploration est assurée grâce à la liste des sous-configurations potentielles qui est calculée dans la fonction *checkRemainings*, or cette fonction utilise aussi le parcours déjà effectué et la composante en cours, pour ne récupérer les sous-configurations potentielles *que* pour les concepts qui n'ont pas encore été visités. Nous sommes donc assurés :

1. de ne jamais visiter deux fois le même concept lors d'un parcours en profondeur ;
2. de ne jamais visiter d'autres concepts que ceux de la composante en cours de test.

Par ailleurs, comme le modèle du domaine n'autorise aucune association réflexive une composante ne peut pas contenir plusieurs fois un même concept. Dès lors, la récursivité se termine toujours lors de notre parcours en profondeur.

Nous pouvons donc garantir la terminaison de l'algorithme de test de la réalisabilité.

6.8.4.2 Complexité

Nous nous appuyons pour définir la complexité de notre algorithme sur la topologie du modèle du domaine. La complexité de l'algorithme va, en effet, dépendre à la fois du modèle du domaine et de la sous-configuration testée.

Soit $\mathcal{M} = \langle SC, SA \rangle$ un modèle du domaine avec $SC = \{\mathcal{D}_0, ..., \mathcal{D}_n\}$ et $SA = \{\mathcal{A}_0, ..., \mathcal{A}_m\}$. Nous pouvons déterminer à partir de $\mathcal{M}$ l'ensemble des composantes biconnexes $SB = \{\mathbb{C}_0, ..., \mathbb{C}_n\}$ en utilisant un algorithme en O($m+n$) avec $m = |SA|$ et $n = |SC|$ [Hopcroft 73]. Nous savons alors que $\forall\ \mathcal{D}_i \in SC$, $\exists SB_i \subseteq SB$ telle que $\forall \mathbb{C}_i \in SB_i$, $\mathcal{D}_i \in \mathbb{C}_i$. Par ailleurs, si $\mathcal{D}_i$ est un point d'articulation, alors $|SB_i| > 1$ sinon $|SB_i| = 1$.

Nous considérons désormais le test d'une unique sous-configuration $\mathcal{SC}_t$ dans une LPL où toutes les sous-configurations ont précédemment été validées : il s'agit de notre hypothèse de récursion.

Soit $\mathcal{SC}_t \rightsquigarrow \mathcal{D}_t$ et SB_t l'ensemble des composantes biconnexes dans lesquelles sont incluses $\mathcal{D}_t$.

Si $|SB_t| = 1$, alors $\mathcal{D}_t$ n'est pas un point d'articulation. Dans ce cas l'algorithme dans le pire des cas pour tester la sous-configuration $\mathcal{SC}_t$ consiste à la comparer avec toutes les sous-configurations de sa composante connexe. En considérant $\mathbb{C}_t$ l'unique composante connexe

de $\mathcal{D}_t$, la complexité, exprimée en nombre de comparaisons dans le pire des cas est donc exactement :

$$\prod_{\mathcal{V}_i \rightsquigarrow \mathcal{D}_i, \mathcal{D}_i \in \mathbb{C}_t} |[\![\mathcal{V}_i]\!]|$$

En considérant :

$$n = \sum_{\mathcal{V}_i \rightsquigarrow \mathcal{D}_i, \mathcal{D}_i \in \mathbb{C}_t} |[\![\mathcal{V}_i]\!]|$$
$$k = |\mathbb{C}_t|$$

nous pouvons alors déterminer que dans ce cas précis la complexité de notre algorithme est de l'ordre O($\frac{n^k}{k}$).

En revanche, si $|SB_t| > 1$ alors $\mathcal{D}_t$ appartient à un point d'articulation et il convient de tester l'ensemble des composantes biconnexes auxquelles il appartient. En considérant $SB_t = \{\mathbb{C}_{t0}, ..., \mathbb{C}_{tk}\}$, la complexité de notre algorithme exprimé en nombre de comparaisons dans le pire des cas est donc exactement :

$$\sum_{i=t0}^{i=tk} (\prod |[\![\mathcal{V}_j]\!]) \text{ pour } \mathcal{V}_j \rightsquigarrow \mathcal{D}_j \text{ avec } \mathcal{D}_j \in \mathbb{C}_i$$

La complexité de notre algorithme est encore une fois de l'ordre O($\frac{n^k}{k}$) mais cette fois en considérant :

$$n = \sum_{i=t0}^{i=tk} (\sum |[\![\mathcal{V}_j]\!]|) \text{ pour } \mathcal{V}_j \rightsquigarrow \mathcal{D}_j \text{ avec } \mathcal{D}_j \in \mathbb{C}_i$$
$$k = \sum_{i=t0}^{i=tk} |\mathbb{C}_i|$$

Ainsi, le pire des cas se produit pour notre algorithme si le modèle du domaine à une topologie de graphe fortement connexe et qu'aucune sous-configuration n'a encore été testée. Dans ce cas, la complexité de l'algorithme se rapproche de celle de l'algorithme naïf. Cependant, si la topologie du modèle du domaine est moins dense, ce qui semble être une hypothèse raisonnable, étant donné qu'un modèle du domaine très dense semble difficilement maintenable, alors notre algorithme est en moyenne plus intéressant : la complexité sera en effet moindre, pour toutes les sous-configurations appartenant à des concepts qui ne sont pas des points d'articulation.

6.8.5 Optimisation par classe d'équivalence

Nous proposons une optimisation à notre algorithme basée sur la notion de classe d'équivalence entre nos différentes sous-configurations. On peut en effet parler de classes d'équivalence dans le cas où plusieurs sous-configurations d'un même concept vont déclencher exactement les mêmes fonctions de restriction : dans ce cas, ces sous-configurations auront exactement les mêmes propriétés de compatibilités avec les sous-configurations des autres concepts.

Il est alors possible de calculer directement les classes d'équivalence d'une LPL, concept par concept, en considérant chacune des règles de restrictions de toutes les fonctions de restriction dont un concept donné est la source. Les états de configurations des différentes règles sont alors autant de prédicats que l'on peut combiner dans une même formule booléenne afin de déterminer

une classe d'équivalence : une sous-configuration appartient alors à une classe d'équivalence si elle respecte exactement cette formule booléenne, ou dit autrement, si elle déclenche exactement ces règles de restriction.

Ainsi, si une sous-configuration appartient à une classe d'équivalence donnée et a été validée, n'importe quelle autre sous-configuration qui appartient à la même classe d'équivalence n'a pas besoin d'être testée : elle sera nécessairement valide. On peut, en effet, remplacer de manière transparente la première sous-configuration testée par la nouvelle sous-configuration sans rien changer à la configuration composite obtenue lors du test via notre algorithme.

6.9 Conclusion

Nous avons ainsi présenté dans ce chapitre les éléments du formalisme et les algorithmes nécessaires à la réalisation d'un processus de configuration flexible et cohérent.

En particulier nous avons formalisé la notion d'action utilisateur et la manière dont ces différentes actions vont travailler avec les contextes de configurations afin de proposer un maximum de flexibilité dans l'ordre des choix utilisateur, tout en lui garantissant la cohérence de ses choix. Nous avons défini afin d'assurer cette cohérence un algorithme de propagation, capable d'assurer la cohérence des différents contextes de configuration tout au long du processus. Par ailleurs nous avons également présenté la notion d'historique de configuration, afin de permettre à l'utilisateur de revenir sur ses actions de configuration, de les annuler à n'importe quel moment tout en garantissant, une fois encore, la cohérence du processus de configuration.

Ainsi il est possible de réaliser une configuration composite dans une LPL complexe en suivant un workflow de configuration extrêmement libre, bien que guidé par les choix utilisateurs et par les informations du domaine.

Enfin, nous avons présenté un algorithme exploitant la topologie du modèle du domaine représentant la LPL afin d'optimiser la vérification de la réalisabilité qui est une propriété essentielle à la cohérence de la ligne.

CHAPITRE 7

UNE IMPLÉMENTATION ÉTENDUE DE L'APPROCHE

Sommaire

Résumé Nous présentons dans ce chapitre une mise en œuvre du formalisme présenté dans les chapitres précédents. L'objectif de ce chapitre n'est pas de donner une description détaillée de l'implémentation mais seulement les éléments qui nous semblent essentiels. Le lecteur intéressé pourra se reporter aux codes disponibles sur GitHub pour plus d'informations [1].

Nous nous intéressons en particulier dans ce chapitre aux éléments constituant une implémentation étendue du formalisme que nous avons défini. Ainsi nous présentons le métamodèle utilisé pour implémenter les différents concepts du formalisme en montrant les choix d'implémentation que nous avons réalisés. Nous présentons ensuite un langage de définition des fonctions de restriction que nous avons créé afin d'aider les architectes de LPL à construire leur modèle du domaine. Nous discutons également de notre méthodologie de développement pour implémenter une solution et nous montrons les différentes API internes et externes que nous avons définies. Enfin nous donnons un aperçu de l'implémentation de l'interface graphique de configuration

1. http://github.com/surli/spinefm

que nous avons réalisée pour permettre à un utilisateur final d'utiliser notre solution au travers d'un processus de configuration flexible.

La contribution de ce chapitre répond en priorité au troisième objectif de cette thèse (voir section 4.4), mais offre également des réponses pratiques aux résultats théoriques obtenus dans les chapitres précédents pour les deux premiers objectifs (voir section 4.2 et section 4.3).

7.1 Introduction

Nous avons réalisé une implémentation complète et étendue du formalisme défini dans les chapitres précédents, nommée SPINEFM [2]. Nous présentons dans ce chapitre non seulement les choix d'implémentation et la méthodologie utilisée pour implémenter notre solution, mais nous décrivons également les différentes réalisations effectuées afin de faciliter son utilisation.

Les objectifs de nos travaux de recherches étaient de permettre la modélisation de LPL complexes et de fournir les outils nécessaires à un processus de configuration qui soit à la fois cohérent et flexible. Nous avons montré par notre formalisme comment atteindre ces objectifs. Nous présentons ici comment l'exploitation d'outils liés à notre formalisme permet d'aller plus loin dans l'accomplissement de nos objectifs.

Nous montrons ainsi dans un premier temps une vue d'ensemble de notre implémentation qui résume les choix d'architectures effectuées (section 7.2). Nous présentons ensuite le métamodèle utilisé pour réaliser l'implémentation du formalisme défini dans les chapitres 5 et 6 (section 7.3). Nous nous intéressons en particulier aux choix d'implémentation que nous avons fait au niveau du métamodèle et en quoi ils diffèrent du formalisme. Nous décrivons dans une troisième section un langage de définition des fonctions de restriction que nous avons créé afin de faciliter la modélisation des fonctions et règles de restrictions par l'architecte de la LPL (section 7.4). Nous discutons ensuite de notre méthodologie de développement pour réaliser SPINEFM et décrivons dans la même section les API internes et externes définies et exploitées (section 7.5). Enfin nous présentons les caractéristiques d'une interface graphique de configuration réalisée afin de permettre un processus de configuration flexible et cohérent menant à une configuration composite (section 7.6).

7.2 Vue d'ensemble

Nous donnons une vue d'ensemble de l'implémentation de notre formalisme, nommée SPINEFM, dans la Figure 7.1.

7.2.1 Architecture

Notre implémentation est constituée de quatre parties principales (représentées par une pastille rouge dans la Figure 7.1) :

1. un métamodèle définissant les concepts formalisés dans les chapitres précédents ;
2. un langage spécifique de description des fonctions de restriction appelé RESTFUNC DSL ;
3. un moteur de raisonnement qui implémente les concepts du formalisme et qui offre une API web pour les exploiter ;
4. une interface graphique générique de configuration, appelée TOCSIN, utilisant cette API et exploitant des métadonnées afin de présenter des informations visuellement pertinentes à l'utilisateur final.

Ces éléments ont été définis de manière modulaire au sein d'une architecture décentralisée et sont utilisés par différents acteurs. La définition de cette architecture provient du besoin que nous avons ressenti lors de la réalisation de l'implémentation de permettre un maximum de réutilisation. Les outils existants permettant aujourd'hui de faire des LPL sont généralement des outils complètement intégrés qui impliquent d'exploiter une unique technologie. Nous voulons

2. Le nom SPINEFM a été imaginé comme une "colonne vertébrale pour FM".

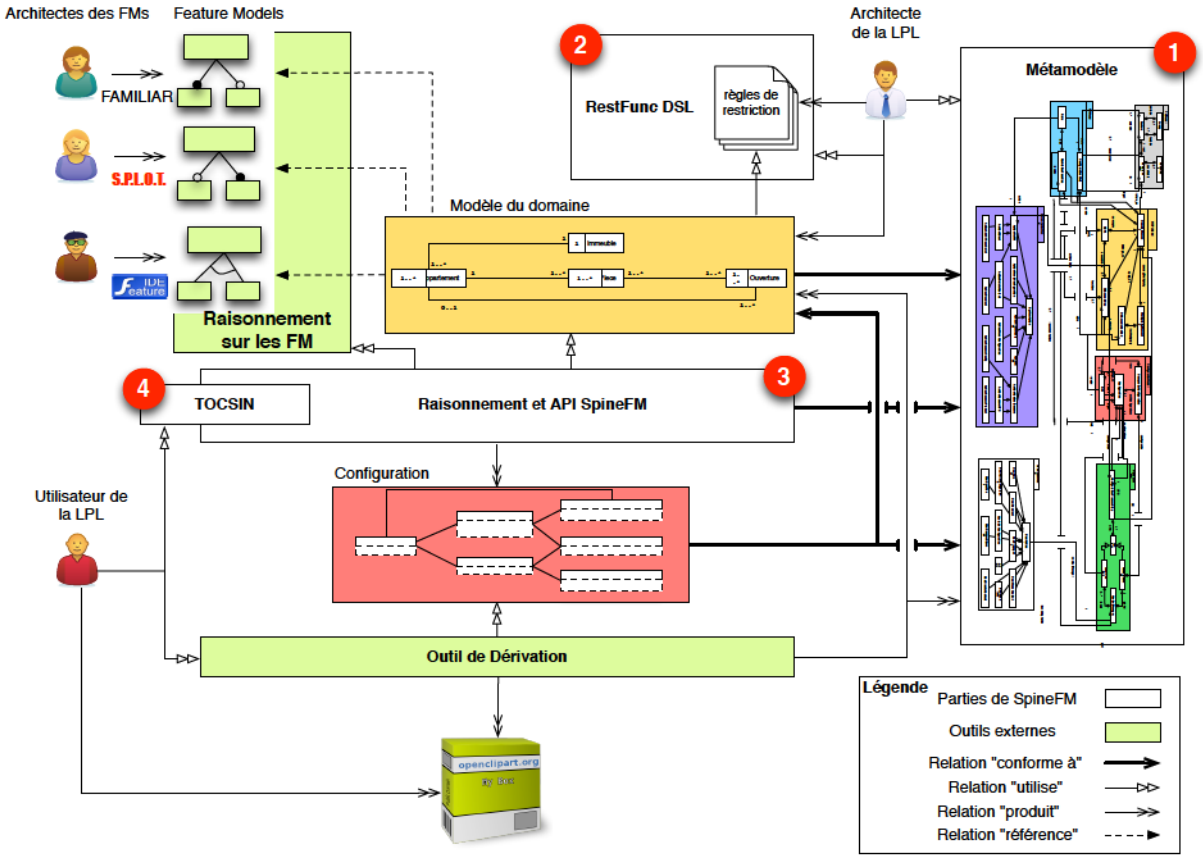


FIGURE 7.1 – Vue d'ensemble de la solution

avec notre approche au contraire favoriser l'exploitation des outils déjà existants de raisonnement sur les FM, de configuration ou encore de génération de code. La définition d'une architecture modulaire avec des API clairement définies nous a semblé une réponse idéale à cette volonté.

7.2.2 Cas d'utilisation et acteurs

En premier lieu un *architecte de la LPL* définit le modèle du domaine. Pour cela, il exploite le métamodèle afin de créer les concepts, associations et multiplicités dont il a besoin. Au sein de son modèle du domaine, il fait référence à des feature models créés par des experts du domaine pour l'occasion ou réutilisés. Finalement il utilise notre langage spécifique, RESTFUNC DSL, afin de définir les différentes fonctions de restriction nécessaires à la LPL.

Dans un second temps, un *utilisateur de la LPL* va pouvoir réaliser un nouveau produit en exploitant nos outils. Pour cela il utilise notre interface de configuration qui communique avec le moteur de raisonnement via une API web : les outils sont définis dans des espaces séparés, chacun répondant à une problématique spécifique.

7.2.3 Fonctionnement et limitations

Le moteur de raisonnement lui-même exploite le formalisme représenté dans le métamodèle ainsi que le langage des fonctions de restriction afin d'utiliser le modèle du domaine et les règles de restrictions précédemment définies. Par ailleurs, il utilise des moteurs de raisonnement sur les FM à travers une API que nous avons définie afin d'utiliser les différents FM référencés par le modèle du domaine. L'interface de configuration exploite les informations du domaine

renvoyées par le moteur de raisonnement, afin de produire un affichage et un guidage qui soient compréhensibles par l'utilisateur. Finalement, une fois la configuration composite terminée, l'utilisateur utilise un outil de dérivation afin de réaliser le produit correspondant à la configuration demandée.

Cette vision d'ensemble nous montre également les limites de notre approche. En particulier, notre approche se base sur des moteurs de raisonnements sur les FM déjà existants. Nous revenons dans la sous-section 7.5.2 sur l'API requise pour exploiter un moteur de raisonnement sur les FM dans SPINEFM. En outre, nous considérons que la fin du processus de dérivation, une fois la configuration obtenue, est intrinsèquement dépendante du domaine métier du produit. Nous présentons dans la validation une implémentation possible, dans un cadre spécifique, d'un outil de dérivation. Nous revenons également sur ce point dans les perspectives de notre approche.

7.3 Métamodèle pour SpineFM

Notre implémentation de SPINEFM a été réalisée en utilisant des outils de l'ingénierie dirigée par les modèles. Nous avons en particulier défini un métamodèle afin de représenter les différents concepts et leurs relations et d'automatiser la génération d'une grande partie du code de SPINEFM. Pour cela nous avons utilisé les outils proposés par Eclipse EMF [3]. Une partie du métamodèle obtenu est visible dans la Figure 7.2 et reprend les éléments du formalisme défini dans les chapitres 5 et 6. Cette figure représente les concepts et une partie des associations de notre métamodèle. Sa définition EMF contient également un ensemble d'opérations prédéfinies que nous n'avons pas représentées dans un souci de simplicité.

Le métamodèle est composé de 8 packages :

1. *FMModel* représente les concepts inhérents aux FM. Cette représentation vise à garantir l'indépendance avec les moteurs de raisonnement sur les FM. Elle se limite donc aux concepts indispensables au raisonnement. Nous nous sommes par ailleurs inspirés du métamodèle réalisé dans le projet EMF Feature Model [4].
2. *SPLModel* représente les concepts relatifs au modèle du domaine ;
3. *ConfigurationModel* représente les concepts utilisés pour la représentation d'une configuration composite ;
4. *RFModel* représente les concepts utilisés pour gérer les règles et fonctions de restriction ;
5. *ProcessModel* représente les concepts utilisés pour la gestion du processus de dérivation ;
6. *SystemActionModel* représente les différentes actions système ;
7. *UserActionModel* représente les différentes actions utilisateur ;
8. *HistoryModel* représente les concepts relatifs à l'historique.

On peut constater que si les concepts du formalisme se retrouvent tous dans le métamodèle, celui-ci en expose davantage. En particulier, des actions système supplémentaires sont définies afin de permettre le nommage des différents éléments : les sous-configurations, CPS et configuration composite peuvent en effet être nommées par l'utilisateur dans notre implémentation. De la même façon, l'ensemble des entités définies dans le métamodèle ont un identifiant unique utilisé lors des opérations. En outre des associations supplémentaires sont également présentes

3. http://www.eclipse.org/modeling/emf/
4. http://www.eclipse.org/featuremodel/

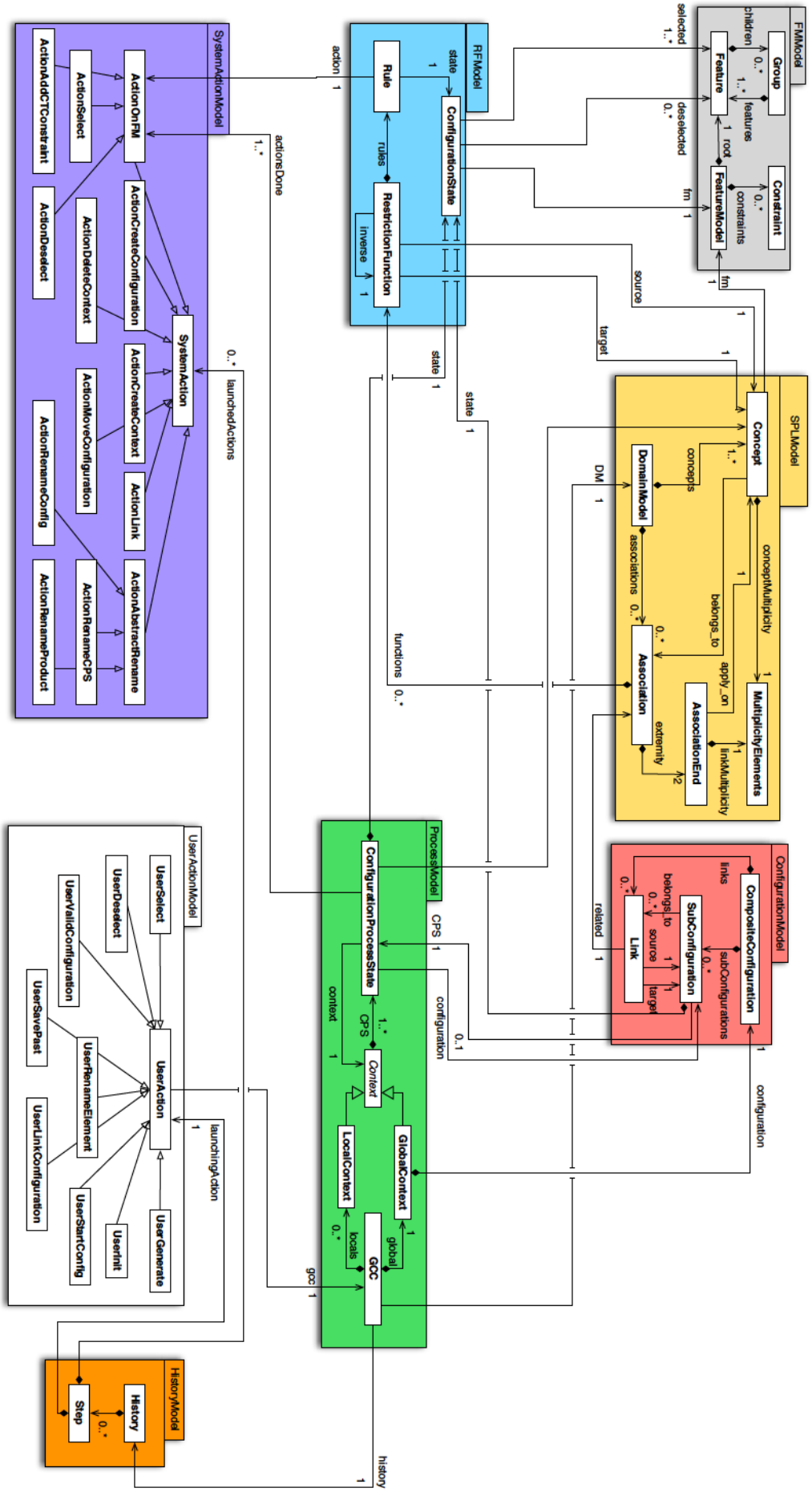


FIGURE 7.2 – Métamodèle développé pour SPINEFM

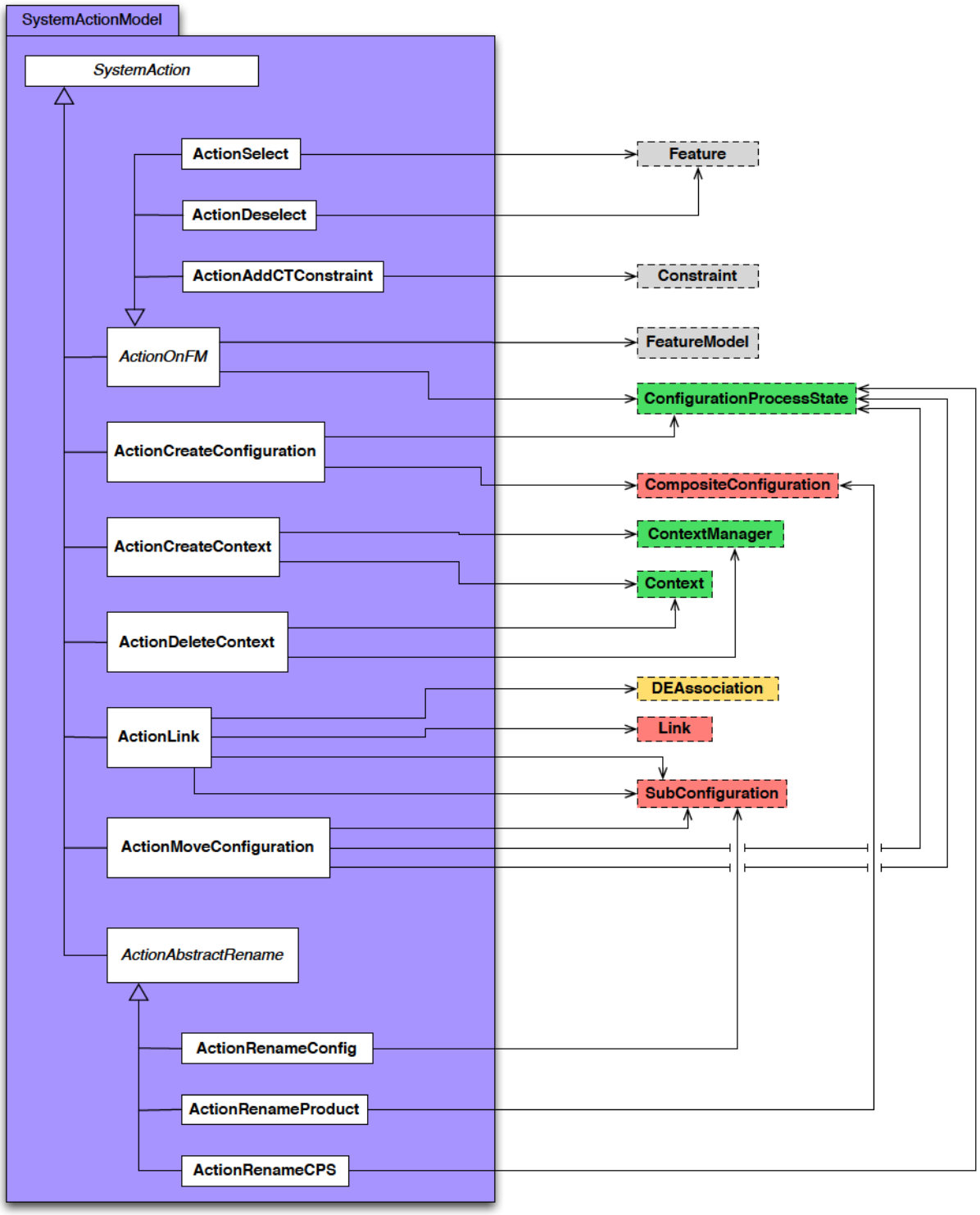


FIGURE 7.3 – Actions système et concepts du métamodèle

dans le métamodèle par rapport à la définition des concepts dans le formalisme : celles-ci ont été ajoutées dans le métamodèle afin de faciliter l'implémentation de SPINEFM.

La Figure 7.2 a par ailleurs été simplifiée afin de ne pas représenter les liens entre les actions système et les différents concepts auxquels elles font référence. Ces informations sont visibles dans la Figure 7.3.

On constate ainsi que les actions système font référence à un grand nombre de concepts

différents : en particulier les ConfigurationProcessStep sont très utilisés dans les actions.

7.4 RestFunc DSL : un langage pour les règles de restriction

Dans l'optique de faciliter le travail de modélisation du domaine, nous avons développé un langage minimaliste permettant de définir rapidement des ensembles de règles de restriction.

Nous avons ainsi défini une grammaire permettant de décrire une fonction de restriction en utilisant les outils proposés par Eclipse Xtext[5]. Pour rappel, une fonction de restriction est orientée d'un concept vers un autre et contient un ensemble de règles de restriction (voir section 5.4).

Listing 7.1 – Grammaire de RestFunc DSL

```
grammar fr.unice.spinefm.RestfuncDSL with org.eclipse.xtext.common.Terminals

generate restfuncDSL "http ://www.unice.fr/spinefm/RestfuncDSL"

// Definition de la fonction
RestFunc : init=SourceAndTarget (rules+=RuleString)+ ;

SourceAndTarget : source=Source target=Target;
Source : SOURCEKEY EQUAL deSrc=ID;
Target : TARGETKEY EQUAL deTarget=ID;

// Definition d'une regle
RuleString : BEGIN_RULE (id=ID) ? POINTS left=LeftPart IMPLY right=RightPart END_RULE;

// partie gauche
LeftPart : (features+=GroupFeature)+ ;
GroupFeature : state=FM_STATE feature+=FeatureNamed (COMMA feature+=FeatureNamed)* ;

// partie droite
RightPart : ActionOnFeature | ActionOnFM ;
ActionOnFeature : action=ACTIONFEATURE feature=FeatureNamed ;
ActionOnFM : action=ACTIONFM feature+=FeatureTrueOrFalse (COMMA feature+=FeatureTrueOrFalse)* ;

// Recuperation d'une feature ou d'une variable
FeatureNamed : SingleFeature | VariableFeature | StarFeature;
SingleFeature : featureName=ID;
VariableFeature : featureName=ID DOT variable=Variable;
Variable : VARBEGIN id=ID;
StarFeature : featureName=ID DOT STAR;
FeatureTrueOrFalse : FeatureNamed | NOT FeatureNamed;

// elements terminaux
FM_STATE : 'SELECTED' | 'DESELECTED';
ACTIONFEATURE : 'SELECT' | 'DESELECT';
ACTIONFM : 'ADDCONSTRAINT';
terminal IMPLY : '=>';
terminal BEGIN_RULE : 'rule';
terminal STAR : '*';
terminal POINTS : ' :';
terminal END_RULE : ';';
terminal COMMA : ',';
terminal DOT : '.';
terminal VARBEGIN : '$';
terminal SOURCEKEY : 'source';
terminal TARGETKEY : 'target';
terminal EQUAL : '=';
terminal NOT : '!';
```

Notre grammaire, représentée par son implémentation Xtext dans le Listing 7.1, permet de préciser les concepts sources et cibles de la fonction, et facilite l'écriture des règles par l'exploitation des noms de features et de la hiérarchie au sein du FM.

5. http://www.eclipse.org/Xtext/

Il est ainsi possible d'exploiter un nom de feature en utilisant les trois syntaxes suivantes :

1. `feature`
2. `feature.*`
3. `feature.$n` où `n` représente un nombre quelconque.

La première syntaxe permet d'accéder simplement à un nom de feature identifié. La seconde syntaxe permet d'accéder à l'ensemble des features enfants. Enfin, la dernière syntaxe permet de créer des variables utilisables en partie droite et en partie gauche.

Exemple 21 : Utilisation de RestFunc DSL

Par exemple, si l'utilisateur souhaite créer différentes règles de restriction à utiliser avec les FM représentés dans la Figure 7.4.

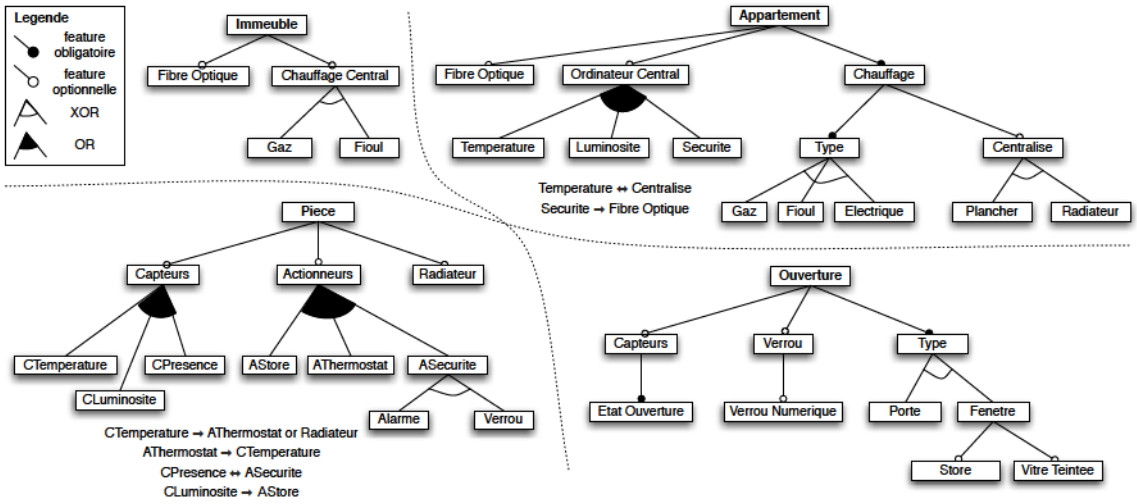


FIGURE 7.4 – Feature Models pour l'exemple fil rouge

Dans le premier cas, l'utilisateur souhaite créer une règle exprimant le fait que s'il a sélectionné la feature *Temperature* dans *Appartement*, alors il devra sélectionner la feature *CTemperature* dans *Piece*.

Dans le langage RestFunc DSL, cela s'exprime par la syntaxe suivante :

Listing 7.2 – Expression d'une règle simple en RestFunc DSL

```
rule appartement_piece_temperature

source  : Appartement
target  : Piece

SELECTED Temperature => SELECT CTemperature ;
```

Nous avons délibérément utilisé dans l'ensemble du document une syntaxe d'expression des règles très proches de la syntaxe de RestFunc DSL.

Cependant, nous pouvons exprimer des règles en exploitant une hiérarchie. Par exemple, si nous voulons exprimer le fait que la sélection de tous les modules de l'*ordinateur central* dans *Appartement* va automatiquement sélectionner la feature *Fibre optique* dans *Immeuble*, nous pouvons utiliser la syntaxe suivante :

Listing 7.3 – Expression d'un ensemble de règles avec *

```
rule appartement_immeuble_ordinateur

source : Appartement
target : Immeuble

SELECTED OrdinateurCentral.* => SELECT Fibre Optique ;
```

Cette syntaxe va automatiquement générer la règle suivante lors de l'initialisation du système :

Listing 7.4 – Génération d'un ensemble de règles avec *

```
rule appartement_immeuble_ordinateur

source : Appartement
target : Immeuble

selected [Temperature, Luminosite, Securite] deselected [] => SELECT Fibre Optique ;
```

Enfin, une dernière syntaxe nous permet de faire de la comparaison de noms de features tout en exploitant une hiérarchie. Par exemple, si l'on souhaite créer toutes les règles permettant de sélectionner le *type de chauffage* d'un *Appartement* à partir du *Chauffage central* d'un *Immeuble* on peut utiliser la syntaxe suivante :

Listing 7.5 – Expression d'un ensemble de règles avec $

```
rule immeuble_appartement_chauffage

source : Immeuble
target : Appartement

SELECTED ChauffageCentral.$1 => SELECT Type.$1 ;
```

Cette syntaxe va automatiquement parcourir les deux hiérarchies sous *Chauffage Central* et sous *Type* et créer les règles pour toutes les features possédant le même nom. On obtient donc les règles suivantes :

Listing 7.6 – Génération d'un ensemble de règles avec $

```
rule immeuble_appartement_chauffage_gaz

source : Immeuble
target : Appartement

selected [Gaz] deselected [] => SELECT Gaz ;

rule immeuble_appartement_chauffage_fioul

source : Immeuble
target : Appartement

selected [Fioul] deselected [] => SELECT Fioul ;
```

7.5 Développement et exposition des services

Le moteur de raisonnement SPINEFM a été implémenté en Java en exploitant une génération automatique des codes réalisée à partir du métamodèle présenté dans la section 7.3 et permise par les outils d'Eclipse EMF. Nous présentons dans cette section la méthodologie de développement utilisée dans le cadre de l'implémentation de SPINEFM. Nous décrivons ensuite l'API de raisonnement sur les FM dont nous avons besoin, puis nous définissons l'API Web réalisée.

7.5.1 Méthodologie de développement de SpineFM

SpineFM a été réalisé de manière incrémentale en exploitant les possibilités offertes par l'ingénierie dirigée par les modèles. En effet, nous avons commencé par définir un métamodèle représentant un ensemble restreint de concepts à exploiter pour réaliser des configurations de LPL complexes. A partir de ce premier métamodèle nous avons généré automatiquement une base de code que nous avons enrichie avec nos propres implémentations, tout en conservant une séparation nette entre le code écrit et le code généré.

Ce faisant nous avons pu faire évoluer le métamodèle afin d'intégrer de nouveaux concepts ou d'en modifier d'autres : à chaque étape nous avons été en mesure de générer une nouvelle base de code et d'adapter l'implémentation écrite qui utilise cette base de code.

En définitive, l'implémentation complète du moteur de raisonnement SPINEFM contient environ 90 000 lignes de code en Java dont la moitié a été automatiquement générée à partir de notre métamodèle.

7.5.2 API de raisonnement sur les FM

Nous avons défini une API de raisonnement sur les FM à implémenter pour chaque moteur de raisonnement que l'on souhaite utiliser avec SPINEFM.

Listing 7.7 – API minimale de raisonnement sur les FM

```
1  package fr.unice.spinefm.fmengine;
2
3  import fr.unice.spinefm.FMModel.Constraint;
4  import fr.unice.spinefm.FMModel.Feature;
5  import fr.unice.spinefm.FMModel.FeatureModel;
6  import fr.unice.spinefm.fmengine.exceptions.FMEngineException;
7
8  public interface FMSpineFMAdapter {
9
10         // FM State
11
12         public Set<Feature> getSelectedFeatures(String confName, FeatureModel fm) throws
               FMEngineException;
13
14         public Set<Feature> getDeselectedFeatures(String confName, FeatureModel fm) throws
               FMEngineException;
15
16         public Set<Feature> getUnselectedFeatures(String confName, FeatureModel fm) throws
               FMEngineException;
17
18         public boolean isValidConfiguration(String confName) throws FMEngineException;
19
20         public Set<Set<Feature>> getAllConfigurations(String confName) throws
               FMEngineException;
21
22         // Actions
23
24         public void selectFeatureInConfiguration(Feature ft, String configuration) throws
               FMEngineException;
25
26         public void unselectFeatureInConfiguration(Feature ft, String configuration) throws
               FMEngineException;
27
28         public void deselectFeatureInConfiguration(Feature ft, String configuration) throws
               FMEngineException;
29
30         public void addConstraint(Constraint c, String confName) throws FMEngineException;
31
32         public void removeConstraint(Constraint c, String confName) throws FMEngineException;
33
34         public void createConfiguration(String confName, FeatureModel refers_on) throws
               FMEngineException;
```

```
35
        // Utils
37
        public void populateFeatureModel(FeatureModel fm) throws FMEngineException;
39
        public boolean isEquivalent(FeatureModel fm1, FeatureModel fm2);
41
        public String getRepresentationOfFeatureModel(FeatureModel fm);
43
        public void destroyConfiguration(String confName) throws FMEngineException;
45
        public void copyElement(String elementSource, String elementTarget) throws
            FMEngineException;
47
        public void exitInterpreter();
49 }
```

Cette API représentée par l'interface donnée dans le Listing 7.7 contient les méthodes de raisonnement sur les FM indispensables au bon fonctionnement de SPINEFM.

Nous définissons dans les lignes 10 à 22 des méthodes de récupération de l'état d'une configuration : ces méthodes sont utilisées pour mettre à jour l'état de configuration des CPS, mais également pour tester la cohérence et la complétude d'une configuration.

Nous explicitons ensuite les différentes méthodes permettant d'appliquer des actions sur les configurations dans les lignes 22 à 35. L'implémentation de ces méthodes doit absolument permettre un mécanisme de raisonnement dynamique (voir section 6.2) au sein de l'outil de raisonnement sur les FM.

Enfin des méthodes utilitaires, définies dans les lignes 36 jusqu'à la fin, sont également requises pour tester l'équivalence de deux FM ou encore pour "peupler" un FM : il s'agit ici de récupérer automatiquement la définition complète d'un FM (features, groupes et contraintes) à partir de la définition du FM au sein du moteur de raisonnement. Cette méthode permet d'éviter à l'architecte de la SPL de devoir redéfinir les FM en utilisant notre métamodèle : il suffit d'en avoir une représentation utilisant le formalisme supporté par le moteur de raisonnement et d'utiliser une convention de nommage pour y accéder. Le "peuplement" du FM permet alors de reconstruire le FM dans notre formalisme en exploitant les concepts de notre métamodèle.

La définition de cette API d'exploitation des moteurs de raisonnement sur les FM nous permet ainsi de conserver une séparation claire au sein de notre implémentation entre SPINEFM et les différentes librairies utilisées pour exploiter les FM.

7.5.3 API Web de SpineFM

Une API Web a été développée au sein de SPINEFM afin conserver une séparation nette entre le raisonnement apporté par SPINEFM, permettant de réaliser un processus flexible et cohérent de configuration, et l'interface de configuration dont la problématique est de représenter la variabilité et de guider l'utilisateur dans ses choix. Par ailleurs, nous postulons que les interfaces de configuration peuvent être très variées selon les domaines, les buts recherchés etc. Définir une API nous permet ainsi d'autoriser l'utilisation de SPINEFM sans être dépendant de notre interface de configuration. Le choix d'une API "Web" provient d'une volonté que notre moteur de raisonnement soit accessible de partout et puisse être interrogé sans nécessiter l'installation d'un logiciel particulier.

Les ressources considérées dans notre API sont le modèle du domaine, les CPS, les sous-configurations et la configuration composite. Notre API permet essentiellement d'effectuer les actions utilisateur du métamodèle (voir Figure 7.2) mais également de récupérer toutes les informations de la LPL qui peuvent être utiles pour l'interface de configuration.

En tant qu'API Web, les différentes ressources sont accessibles par l'utilisation de différents chemins, chaque chemin correspondant à un appel de fonction précis. Nous présentons ainsi dans le Tableau 7.1 les fonctionnalités de notre API à travers ces différents chemins.

Chemin	Description et retour
Initialisation	
/	Lance l'action d'initialisation. Permet de commencer à réaliser une nouvelle configuration composite. Retourne un token identifiant cette nouvelle configuration. Nous l'appelons **token** dans la suite.
/[token]/name/[nom]	Associe un nom à la configuration composite. Retourne un booléen.
/[token]/name/	Récupère le nom de la configuration composite. Retourne une chaîne de caractères.
Modèle du domaine	
/[token]/model/	Récupère les informations du modèle du domaine : concepts, associations et multiplicités. Retourne une réponse structurée.
/[token]/model/concepts	Récupère la liste des concepts du modèle du domaine. Retourne une liste.
/[token]/model/[concept]/multiplicity	Récupère la multiplicité d'un concept donné. Retourne une réponse structurée.
/[token]/model/[concept]/linkables	Récupère la liste des concepts associés à un concept donné. Retourne une liste.
/[token]/model/asso/src/[concept]/target/[concept2]/multiplicity	Récupère la multiplicité d'une association entre les deux concepts donnés, du côté du concept cible. Retourne une réponse structurée.
CPS	
/[token]/cps/global/[concept]	Récupère l'état de configuration d'un concept au sein du contexte global. Retourne un arbre de feature correspondant au FM dont chaque feature possède un état "sélectionné", "exclu" ou "non sélectionné".
/[token]/cps/[contexte]/[concept]	Récupère l'état de configuration d'un concept au sein du *contexte donné*. Retourne un arbre de feature correspondant au FM dont chaque feature possède un état "sélectionné", "exclu" ou "non sélectionné".
/[token]/cps/[contexte]/[concept]/isValid	Détermine si l'état de configuration d'un concept au sein du *contexte donné* est complet (voir Propriété 5.3.2). Retourne un booléen.
/[token]/cps/[contexte]/[concept]/select/[feature]	Sélectionne la feature d'un concept donné dans un contexte donné. Retourne un booléen.
/[token]/cps/[contexte]/[concept]/exclude/[feature]	Exclut la feature d'un concept donné dans un contexte donné. Retourne un booléen.
/[token]/cps/[contexte]/[concept]/name/[nom]	Associe un nom au CPS d'un concept donné dans un contexte donné. Retourne un booléen.

Chemin	Description et retour
/[token]/cps/[contexte]/[concept]/name	Récupère le nom d'un CPS d'un concept donné dans un contexte donné. Retourne une chaîne de caractères.
Sous-configuration	
/[token]/config/start/[concept]/[ctx]/[linkedConcept]	Démarre une nouvelle sous-configuration dans un contexte donné dans l'objectif de la lier avec le second concept donné. Retourne l'identifiant du contexte contenant le CPS de la nouvelle sous-configuration.
/[token]/config/valid/[concept]/[ctx]	Valide la sous-configuration du concept donné dans le contexte donné. Retourne l'identifiant de la sous-configuration validée.
/[token]/config/[configID]/isLinked	Détermine si l'ensemble des liens de cette sous-configuration ont été réalisés d'après les associations et les multiplicités de son concept. Retourne un booléen.
/[token]/config/[configID]/isLinkableWith/[concept]	Vérifie si les associations et multiplicités peuvent autoriser la création d'un lien entre une sous-configuration du concept donné et la sous-configuration donnée. Retourne un booléen.
/[token]/config/[configID]/compliant/[concept]	Recherche l'ensemble des sous-configurations du concept donné compatibles avec la sous-configuration donnée. Retourne une liste d'identifiants de sous-configurations compatibles.
/[token]/config/[configID]/linked/[concept]	Recherche l'ensemble des sous-configurations du concept donné liées avec la sous-configuration donnée. Retourne une liste d'identifiants de sous-configurations liées.
/[token]/config/[configID]/name/[nom]	Associe un nom à une sous-configuration. Retourne un booléen.
/[token]/config/[configID]/name	Récupère un nom de sous-configuration. Retourne une chaîne de caractères.
/[token]/link/create/src/[configID]/target/[configID2]	Crée un lien entre les deux sous-configurations données. Retourne une réponse élaborée contenant les informations de modification de contextes : contexte supprimé et/ou créé.
Configuration composite	
/[token]/config/status	Recherche l'ensemble des informations sur les sous-configurations validées. Retourne une réponse contenant l'ensemble des informations agrégées.
/[token]/config/links	Recherche l'ensemble des informations sur les liens créés. Retourne une réponse contenant l'ensemble des informations agrégées.

Chemin	Description et retour
/[token]/config/isValid	Détermine si la configuration composite est terminée. Retourne un booléen.
Annulation d'une action	
/[token]/history	Récupère la liste des actions de l'utilisateur. Retourne une liste structurée contenant les identifiants des étapes et la description des actions.
/[token]/history/undo	Annule la dernière action de l'utilisateur.
/[token]/history/undo/[StepID]	Annule l'action de l'utilisateur sauvée à l'étape donnée.
Finalisation	
/[token]/savePast	Sérialise l'ensemble des actions utilisateur réalisées jusqu'à présent. Retourne un chemin vers le fichier.
/[token]/finalize	Sérialise la configuration composite. Retourne un chemin vers le fichier.

TABLE 7.1 – Description de l'API Web

Nous avons ainsi une API complète permettant l'exploitation de SPINEFM à travers n'importe quelle interface.

7.6 TOCSIN : une interface graphique de configuration pour SpineFM

Nous avons réalisé dans le cadre de l'implémentation de notre approche une interface graphique de configuration exploitant SPINEFM, appelée TOCSIN [6]. Cette interface a été réalisée en utilisant le framework javascript AngularJS [7].

TOCSIN vise à fournir une interface de pilotage générique de SPINEFM afin de réaliser le processus de configuration. Cela implique à la fois de présenter à l'utilisateur la variabilité de manière intelligible mais également de lui fournir un minimum de guidage pour que l'utilisateur sache où il en est dans le processus de configuration, tout en lui laissant le maximum de liberté offert par SPINEFM.

Dans l'objectif de rendre TOCSIN le plus générique possible, nous avons décidé d'exploiter au maximum les informations du modèle du domaine accessibles à travers SPINEFM. En effet, ces informations peuvent être exploitées pour le guidage utilisateur : elles déterminent les concepts liés, les sous-configurations qu'il est nécessaire de réaliser pour chacun des concepts etc. Par ailleurs, il nous est apparu que représenter les FM tels qu'ils sont décrits, c'est-à-dire sous la forme d'un arbre parfois très complexe, pouvait être contre-productif pour la réalisation d'une configuration.

Nous avons donc défini un mécanisme permettant d'ajouter des informations supplémentaires au niveau de l'interface de configuration, afin d'enrichir les FM pour faciliter leur compréhension et leur configuration.

6. TOCSIN était un des premiers acronymes envisagé pour le moteur de raisonnement signifiant "TOol for Configuring Software product lINe" ; le nom est resté pour l'interface de configuration.

7. https://angularjs.org/

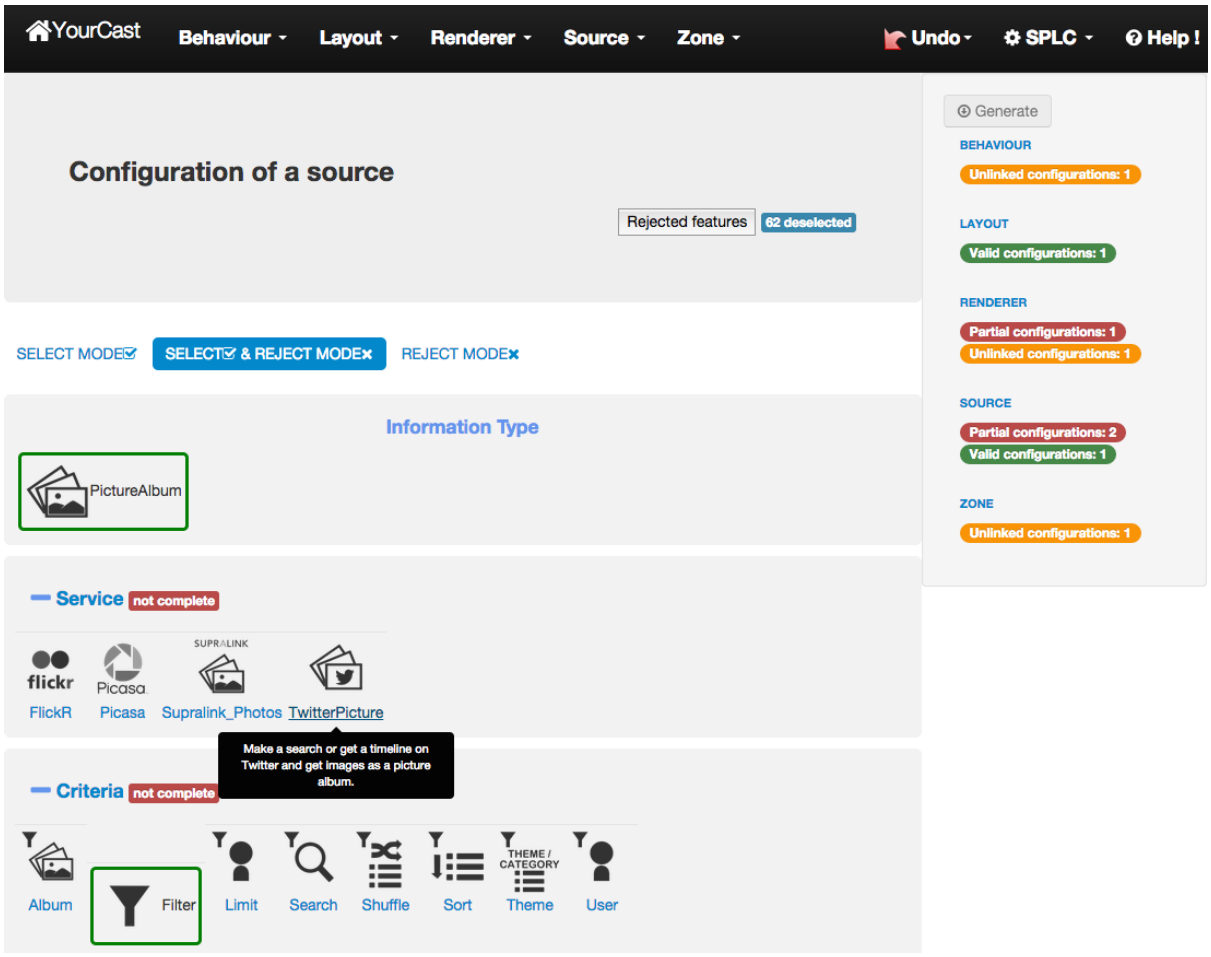


FIGURE 7.5 – Capture d'écran de TOCSIN pour la LPL YOURCAST

7.6.1 FM et métadonnées

Les features d'un FM peuvent ainsi être représentées de manière groupée, par catégories, pour l'utilisateur final. Ces catégories peuvent alors être dérivées de la hiérarchie originelle du FM, ou au contraire être une toute nouvelle organisation, plus parlante pour l'utilisateur. En outre, il nous paraît indispensable qu'aux features puissent être associées des images et des explications, afin d'aider l'utilisateur dans ses choix. De même, les catégories doivent pouvoir bénéficier d'une description afin d'être explicite aux utilisateurs. Enfin, afficher en permanence l'ensemble des catégories n'est pas forcément pertinent pour l'utilisateur : il est donc important de lui donner la possibilité d'ouvrir et fermer les catégories qui l'intéressent.

Cependant il est important de conserver ces informations séparées du FM, la préoccupation étant ici la présentation de l'information, alors qu'un FM vise à fournir une modélisation formelle de la variabilité. Nous avons ainsi développé la notion de métadonnées associées aux FM et réalisé une implémentation au sein de TOCSIN en structurant ces métadonnées au sein d'un fichier JSON.

Listing 7.8 – Exemple de fichier d'annotation pour le FM Appartement

```
1 {
2   "concept" : "Appartement",
3   "description" : "Fichier d'annotation Appartement du 10 Octobre 2014",
4   "accordionBehaviour" : "normal",
5   "zones" : [
6     {
```

```
      "accordion" : false,
      "description" : "Le type de chauffage de l'appartement",
      "level" : 1,
      "name" : "Type de chauffage"
    },
    {
      "accordion" : true,
      "descriptions" : "Options generales",
      "level" : 2,
      "name" : "Options"
    },
    {
      "accordion" : true,
      "description" : "Options de l'ordinateur",
      "level" : 3,
      "name" : "Ordinateur Central"
    },
    {
      "accordion" : true,
      "description" : "Options de chauffage centralise",
      "level" : 4,
      "name" : "Chauffage centralise"
    }
  ],
  "rulesRecursive" : [
    {
      "explication" : "Type de chauffage",
      "feature" : "Type",
      "getChildren" : true,
      "level" : 1,
      "treeLevel" : 1
    },
    {
      "explication" : "Une option de l'ordinateur central",
      "feature" : "Ordinateur Central",
      "getChildren" : true,
      "level" : 4,
      "treeLevel" : 1,
      "logo" : "images/logo/optionOrdinateur.png"
    },
    {
      "explication" : "Mode de chauffage central",
      "feature" : "Centralise",
      "getChildren" : true,
      "level" : 3,
      "treeLevel" : 1
    }
  ],
  "rulesFeatureUnique" : [
    {
      "explication" : "Souhaitez-vous un branchement fibre optique dans votre appartement ?",
      "feature" : "Fibre Optique",
      "level" : 2,
      "logo" : "images/logos/fibre.png"
    },
    {
      "explication" : "Souhaitez-vous un ordinateur centralise dans votre appartement ?",
      "feature" : "Ordinateur Central",
      "level" : 2,
      "logo" : "images/logos/computer.png"
    },
    {
      "explication" : "Souhaitez-vous un chauffage centralise dans votre appartement ?",
      "feature" : "Centralise",
      "level" : 2,
      "logo" : "images/logos/chaudiere.png"
    },
    {
      "explication" : "Souhaitez-vous un appartement equipe de radiateurs ?",
      "feature" : "Radiateur",
      "level" : 3,
      "logo" : "images/logos/radiateur.png"
    }
```

```
80     ]
81 }
```

Un fichier d'annotation JSON contient les métadonnées d'un unique FM, un exemple est donné dans le Listing 7.8. Le fichier contient des informations générales, des informations pour chacune des catégories souhaitées (*zones*, lignes 5 à 30) , puis les annotations d'abord génériques (*rulesRecursive*, lignes 31 à 54) et ensuite spécialisées pour les features (*rulesFeatureUnique*, lignes 55 à 80).

7.6.1.1 Informations générales

Les informations générales du fichier d'annotation contiennent les éléments suivants :
- *description* : une description du fichier d'annotation ;
- *concept* : le nom du concept sur lequel porte le fichier d'annotation ;
- *accordionBehaviour* : le comportement général souhaité pour l'ouverture et la fermeture des catégories. La valeur peut être "normal" ou "openOne" : "normal" signifie que les différentes catégories peuvent être ouvertes et fermées indépendamment les unes des autres ; "openOne" signifie qu'une seule catégorie à la fois peut être ouverte.

7.6.1.2 Informations de catégories

Le fichier d'annotation définit les différentes catégories ou zones qui seront affichées. Chaque zone est définie par les éléments suivants :
- *name* : le nom de la catégorie ;
- *description* : une description plus complète de la catégorie ;
- *accordion* : une valeur booléenne définissant si la catégorie peut être fermée (true) ou si elle est ouverte en permanence (false) ;
- *level* : un nombre définissant la disposition de la catégorie par rapport aux autres : les zones sont affichées par level croissant.

7.6.1.3 Annotations génériques de features

Les annotations génériques sont des annotations qui se définissent pour un ensemble de features. Ces features sont obtenues en parcourant la hiérarchie du FM.

Une annotation générique est définie par les éléments suivants :
- *explication* (facultatif) : définit une explication à appliquer sur l'ensemble des features récupérées à partir de l'annotation ;
- *logo* (facultatif) : définit le chemin vers un logo à afficher pour l'ensemble des features récupérées à partir de l'annotation ;
- *level* : définit la catégorie dans laquelle afficher l'ensemble des features de l'annotation, en utilisant le level comme identifiant ;
- *feature* : définit le nom d'une feature à partir de laquelle constituer l'ensemble des features sur lesquelles appliquer l'annotation. Cette feature est exclue de l'annotation.
- *getChildren* : booléen déterminant si l'ensemble des features héritées de *feature* doit être sélectionné ou non : si une valeur *true* est donnée, l'ensemble du sous-arbre est récupéré.
- *treeLevel* (uniquement utilisé si *getChildren* vaut *false*) : entier déterminant le niveau d'héritage à explorer sous *feature* pour récupérer l'ensemble des features.

Il est ainsi possible en combinant *feature*, *getChildren* et *treeLevel* d'agréger spécifiquement les features d'un niveau du FM. Par exemple, dans le cas du FM *Appartement* si on spécifie *feature : Chauffage*, *getChildren : false* et *treeLevel : 2* alors l'ensemble des features sur lesquelles l'annotation s'applique sera : { Gaz, Fioul, Electrique, Plancher, Radiateur }.

Par ailleurs, plusieurs annotations génériques peuvent annoter les features avec le même level : les features seront alors affichées dans la même catégorie.

7.6.1.4 Annotations spécifiques de features

Les annotations spécifiques sont des annotations réalisées pour une feature précise.

Une annotation spécifique est définie par les éléments suivants :

— *feature* : la feature sur laquelle s'applique la règle spécifique ;

— *level* : la catégorie dans laquelle afficher la feature, en se basant sur le level ;

— *explication* : une explication précise du rôle de la feature ;

— *logo* : un chemin vers un logo à afficher pour cette feature.

Si une annotation générique et une annotation spécifique conviennent à une feature, alors celle-ci sera d'abord annotée par l'annotation générique, puis par l'annotation spécifique. Les informations de l'annotation spécifique sont donc toujours prioritaires sur celles de l'annotation générique.

7.6.2 Actions et guidages

TOCSIN se veut une interface de configuration générique exploitable pour n'importe quelle ligne de produits logiciels définie dans SPINEFM. L'utilisateur spécifie à l'initialisation quel modèle du domaine il souhaite utiliser afin de créer un nouveau produit : l'interface s'initialise alors en récupérant d'un côté les informations à partir de SPINEFM et de l'autre en chargeant les fichiers d'annotations disponibles pour le modèle donné.

L'interface est ensuite extrêmement libre : l'utilisateur a la possibilité de commencer la configuration à partir de n'importe quel concept, et un menu est en permanence présent pour commencer une autre sous-configuration d'un autre concept ou revenir sur une sous-configuration partielle ou terminée. En outre, les écrans de validation des sous-configurations proposent à l'utilisateur un guidage basé sur les associations du modèle du domaine. Par exemple, si un utilisateur valide une sous-configuration d'appartement, l'interface proposera automatiquement la liste des sous-configurations de *Piece* terminées et compatibles, ou alors de créer une nouvelle sous-configuration de *Piece* à lier avec celle qui vient d'être validée.

7.7 Conclusion

Nous avons défini une implémentation étendue de notre formalisme. Cette implémentation est le résultat d'un processus de développement itératif exploitant des outils de l'ingénierie dirigée par les modèles, ce qui nous a permis de conserver un haut niveau d'abstraction pour la conception de notre solution.

Par ailleurs, l'implémentation d'une solution permettant la définition d'une LPL complexe et son exploitation au travers d'un processus de configuration cohérent et flexible ne peut se limiter selon nous à la seule implémentation du formalisme précédemment défini. Nous postulons en effet que pour être utilisé correctement il est nécessaire d'avoir à sa disposition les outils additionnels que nous avons réalisés : un langage dédié permettant de définir facilement des

règles de restriction, une API permettant d'exploiter des moteurs de raisonnement sur les FM, une interface de configuration générique permettant de réaliser le processus de configuration.

Enfin, parlant d'un framework de définition et d'exploitation de LPL, il nous semblait dommage de fournir une solution complètement intégrée offrant peu de capacité de flexibilité et d'évolution. Nous avons donc implémenté notre solution en offrant une architecture décentralisée et des API bien définies afin d'adapter facilement notre solution à différents contextes d'utilisation.

Troisième partie

Validation

CHAPITRE 8

MODÉLISER LA LPL YOURCAST

Sommaire

Résumé Nous validons la première partie de notre approche relative à la modélisation de la variabilité d'une LPL complexe, par la réalisation d'une ligne de produits de portée industrielle dédiée à la création de Systèmes de Diffusion d'Informations (SDI).

Nous montrons dans ce chapitre les éléments de variabilité et le fonctionnement des SDI que nous souhaitons manipuler et décrivons la LPL que nous avons modélisée pour des SDI. Nous discutons ensuite la validation de nos objectifs de modélisation de la variabilité des LPL complexes par l'analyse des résultats obtenus.

8.1 Introduction

Nous présentons dans ce chapitre une validation de nos travaux sur un cas d'utilisation de portée industrielle dans le domaine des Systèmes de Diffusion d'Informations (SDI).

Nous montrons ainsi dans un premier temps en quoi les SDI présentent une large variabilité et dans quelle mesure il est intéressant de réaliser une LPL afin de pouvoir définir des produits. Nous revenons également brièvement sur l'historique du projet *YourCast* dont l'objectif était de réaliser une LPL pour SDI.

Nous présentons dans un second temps la plateforme technologique sur laquelle nous nous sommes appuyés pour définir les éléments communs de notre famille de produits, ainsi que l'architecture d'un SDI dans notre mise en œuvre. A partir de ces éléments, nous définissons le modèle du domaine et décrivons l'approche que nous avons suivie afin de définir les différents FM.

Finalement nous discutons la validation de notre objectif relatif à la modélisation de la variabilité dans les LPL complexes.

8.2 Systèmes de Diffusion d'Informations et Variabilité

Nous définissons tout d'abord la notion de Système de Diffusion d'Informations avant de revenir sur l'historique du projet YOURCAST et de présenter la variabilité de ce type de système.

8.2.1 Définition

Un Système de Diffusion d'Information (SDI) regroupe l'ensemble des éléments nécessaires au bon fonctionnement d'un écran de diffusion d'information.

Appelés parfois "le 5ème écran" – les 4 premiers étant l'écran de cinéma, de télévision, d'ordinateur et de smartphone –, l'écran de diffusion d'information se répand depuis quelques années de manière exponentielle, entraîné par la diminution drastique des coûts sur les matériels électroniques [Kelsen 10]. On retrouve ainsi ce type d'écrans dans de très nombreux lieux et organisations aussi bien publiques (arrêts de bus, universités, mobilier urbain, gares, aéroports, etc) que privés (bureaux d'entreprises, salles d'attentes, particuliers, etc).

Si le rôle de ces écrans est avant tout de donner de l'information, la multiplicité des contextes d'utilisation des écrans donne lieu à de très nombreuses variantes d'utilisation. Il peut ainsi s'agir d'écrans publicitaires, comme on peut en voir dans les galeries marchandes, ou d'écrans d'attente et de divertissement dans les salles d'attentes, ou bien encore d'écrans purement informatifs dans les gares et les aéroports, voire même d'écrans de diffusion interactifs dans certains grands événements, comme les festivals ou les salons. Le plus souvent, cependant, les écrans d'informations sont une combinaison de tous ces usages réalisée pour un contexte et un emplacement particulier.

> "*With a combination of dynamic content that is highly adaptable to its audience and environment, high-definition video imagery, and often interactive features, digital signage can help us, motivate us, provoke us and unite us.*
>
> [KELSEN 10]

(a) Ecran diffusé aux Choralies 2013 (b) Ecran de diffusion de l'équipe GLC

FIGURE 8.1 – Exemples d'écrans de diffusion

Ainsi, avec une telle abondance d'écrans aux multiples usages, il existe une très grande famille de produits de SDI, l'écran n'étant que la partie visible d'un tel système.

8.2.2 Historique du projet YourCast

YOURCAST [1] est un projet ANR Emergence [2] démarré en janvier 2011 et terminé en décembre 2014, dont l'objectif était la réalisation d'une LPL destinée à la réalisation de SDI sur mesure pour des personnes non expertes. Nous nous sommes concentrés dans le cadre de ce projet sur une sous-partie de la famille de produits des SDI. Le *scoping* de notre ligne a été réalisé de manière à cibler les SDI à destination des universités, conférences, événements culturels et organismes à destination des personnes présentant des handicaps visuels.

Une partie de ce choix de scoping est historique. En effet, le projet YOURCAST est l'aboutissement de plusieurs années d'expérimentations et de prototypages de SDI. Une première solution nommée JSEDUITE avait été réalisée et déployée sur le campus de l'école Polytech'Nice mais également dans deux instituts niçois pour personnes présentant des handicaps visuels (IRSAM et Clément Ader).

Le premier objectif du projet YOURCAST était ainsi de réaliser une LPL permettant de créer des produits offrant les mêmes fonctionnalités que la solution JSEDUITE utilisée à l'origine et utilisable par différents profils tels que des personnels administratifs ou des organisateurs d'événements. Le deuxième objectif était d'augmenter la variabilité de cette première ligne pour accéder à davantage de produits. Par ailleurs, nous avons bénéficié dans le cadre du projet d'un partenaire porteur d'un nouveau terrain d'expérimentation qui a pris la forme d'un festival de 4 000 personnes (les Choralies 2013 de Vaison-La-Romaine) nécessitant l'implantation d'un SDI.

Au cours des trois années du projet nous avons ainsi réalisé de nombreux déploiements en exploitant les possibilités de la LPL que nous avons réalisée et que nous décrivons dans la suite.

8.2.3 Une variabilité complexe

La variabilité d'un SDI peut être vue à deux niveaux principaux : *(i)* la variabilité des informations à diffuser et *(ii)* la variabilité des manières de présenter et diffuser une information.

1. http://www.yourcast.fr
2. ANR-2011-EMMA-013-01

> *Digital signage is as diverse as the content that is displayed. It is technology that meets viewers in their environment with potentially the right message at the right time.*
>
> [KELSEN 10]

8.2.3.1 Choix des informations à diffuser

La variabilité du choix des informations à diffuser est double : elle concerne à la fois le *type d'information* et la *provenance de l'information*.

Le *type d'information* constitue le format de l'information. Il peut, par exemple, s'agir de textes, de photos, de vidéos ou de types d'informations plus complexes combinant ces différents éléments comme des articles ou des calendriers. Les formats d'information sont virtuellement infinis, cependant dans le cadre d'une famille de SDI nous pouvons limiter les types d'informations à ceux que nous connaissons et savons diffuser.

La *provenance de l'information* est définie par deux notions : la source de l'information et les éléments de filtrage ou leur absence. La source de l'information peut être extrêmement variée : si le SDI est également un CMS [3] alors la source est généralement le CMS du SDI. Mais il peut également s'agir de n'importe quel autre service capable de fournir une information à diffuser, comme un flux RSS disponible sur internet, un fichier sur un serveur, les données d'un capteur, etc. En outre, la source d'information n'est généralement pas suffisante en soi pour savoir quelle information diffuser. Il est nécessaire de préciser des paramètres de filtrage de l'information, *(i)* les cinq derniers éléments d'un flux RSS disponible à une adresse spécifique, *(ii)* le fichier répondant à une expression régulière particulière disponible sur un serveur spécifique, ou *(iii)* l'ensemble des données renvoyées par un capteur adressable à une adresse donnée, etc.

Ainsi la variabilité concernant les choix d'informations est, elle aussi, virtuellement infinie : elle dépend uniquement de l'imagination et des ressources à disposition. Dans le cadre de YOURCAST nous avons restreint cette variabilité en nous focalisant sur les besoins clients immédiats durant la phase de scoping.

8.2.3.2 Choix de la présentation de l'information

La variabilité est également très importante dans la manière de présenter l'information.

Les écrans de diffusion ont tous leur propre *design* avec une charte graphique spécifique et un *layout* décrivant la manière dont sont disposées les différentes *zones* de diffusion des informations, chacune ayant des propriétés spécifiques comme un titre ou un logo.

Par ailleurs, les informations à diffuser peuvent être présentées de différentes manières selon leur nature : un album d'images peut, par exemple, être présenté sous la forme d'une mosaïque d'images ou image par image sous la forme de diapositives. En outre, les *transitions* entre les informations peuvent être animées, sous la forme d'un fondu ou d'un défilement par exemple.

8.2.3.3 Combinaisons et contraintes

Les éléments de variabilité que nous avons présentés dans les sections précédentes peuvent se combiner entre eux. La variabilité des SDI est donc extrêmement importante et complexe à

3. Content Management System : système de gestion de contenus.

représenter et à maintenir.

En outre, des propriétés de cohérence entre ces différents éléments de variabilités doivent être vérifiées : il ne fait pas de sens, par exemple, d'utiliser un mécanisme de mosaïque d'images avec un service ne renvoyant que de l'information textuelle. De même, une zone très petite en hauteur, destinée à accueillir de l'information en continu, ne devrait pas pouvoir afficher des photos. Et à l'inverse, un mécanisme capable de synthétiser vocalement du texte, ne devrait pas être utilisé sur une zone destinée à accueillir uniquement des images.

Nous montrons dans la section suivante les choix que nous avons faits pour représenter la variabilité des SDI tout en assurant leur cohérence.

8.3 Plateforme et modélisation de la variabilité

La LPL dédiée à la réalisation de SDI a été créée en effectuant deux activités en parallèle : le développement de plusieurs produits et la modélisation de la variabilité. Ces deux activités nous ont permis à la fois de déterminer les éléments communs et variants que nous souhaitons pouvoir manipuler de manière automatisée, mais elle a également permis de spécifier l'architecture et la plateforme commune des produits finaux.

8.3.1 Architecture et fonctionnement d'un SDI

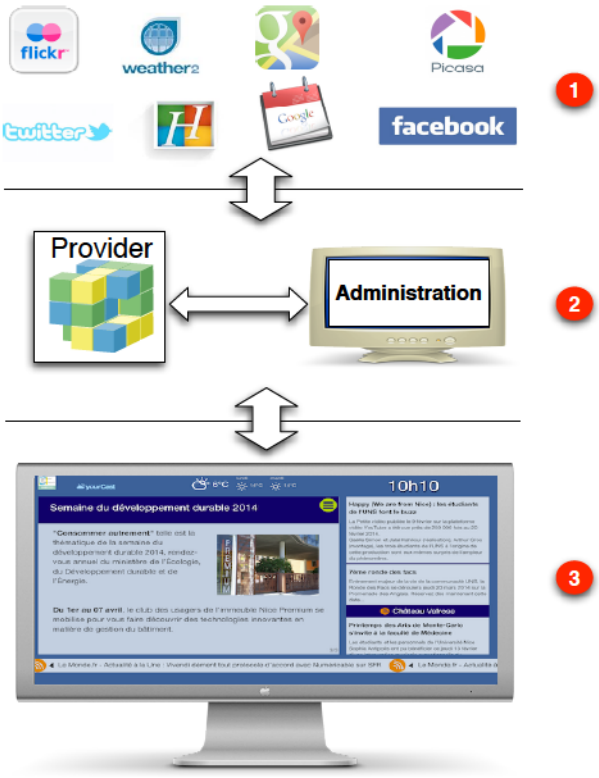


FIGURE 8.2 – Représentation de l'architecture de YourCast

Nous avons défini une architecture orientée services en trois couches pour la réalisation d'un SDI, représentée dans la Figure 8.2 :

1. une première couche contient l'ensemble des *sources* d'information que l'on souhaite pouvoir diffuser ;

2. une seconde couche définit les deux types de services internes utilisés par le SDI : *(i)* une *administration* pour paramétrer les sources (par exemple spécifier le nom d'un compte Twitter à suivre ou un album photo FlickR à récupérer) et *(ii)* des *providers* pour transformer et agréger les informations, zone par zone ;
3. enfin la troisième couche représente le *client* défini par une page web qui récupére les informations à partir des *providers* et les transforme pour les afficher.

Tous les éléments des couches serveurs (couches 1 et 2) ont été développés dans le langage Java et déployés sur un serveur d'application Tomcat [4]. La persistance des données nécessaire à l'interface d'administration a été assurée par une base de données MongoDB [5].

Enfin, concernant le client, il a été décidé de n'employer que des technologies standards permettant d'afficher directement l'écran de diffusion au sein d'un navigateur web. Les technologies JavaScript, CSS, et HTML ont donc été utilisées pour réaliser le *client*.

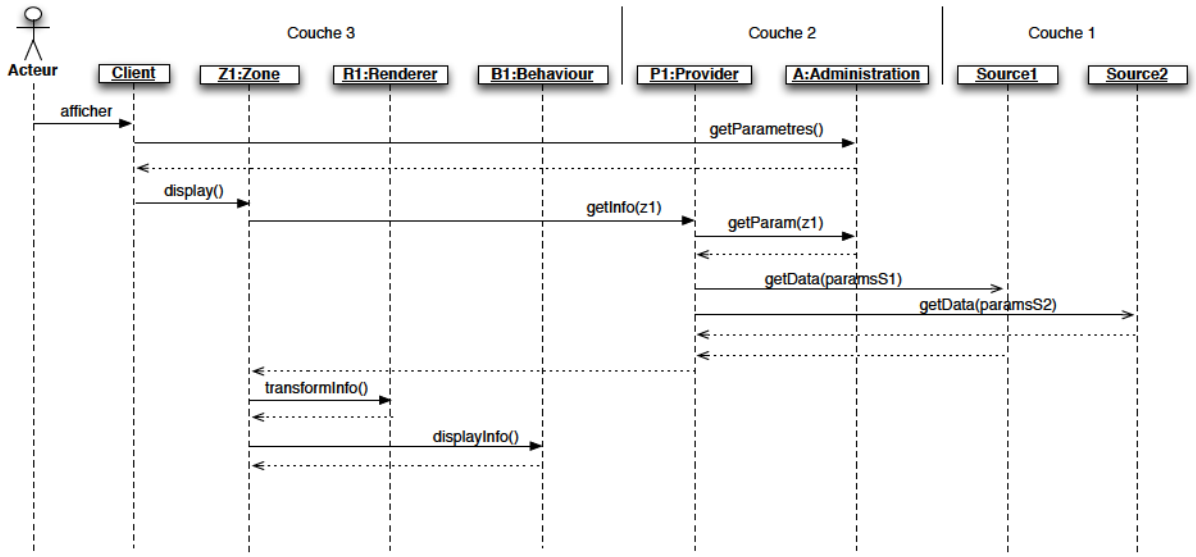


FIGURE 8.3 – Extrait de diagramme de séquence du fonctionnement de YourCast

Le fonctionnement d'un SDI dans YourCast est partiellement représenté dans le diagramme de séquence de la Figure 8.3. Cette figure ne représente l'affichage que d'une unique information. Nous décrivons la séquence permettant cet affichage.

Le *client* de diffusion d'information contient les informations sur les différentes *zones* de diffusion d'information. Ces zones sont indépendantes et sont caractérisées par les types de contenu qu'elles sont susceptibles d'afficher (photo, vidéo, texte, etc) ainsi que par les éléments d'affichage qu'elles peuvent contenir (logo, titre, etc). Lorsqu'un utilisateur demande à afficher une page, le client commence par interroger le *service d'administration* dédié afin de récupérer différents paramètres tels que l'ordre d'affichage des informations, les délais d'affichage etc, ces informations étant utilisées au sein de chaque zone.

Le client demande ensuite à chaque zone d'afficher des informations. Dans la Figure 8.3 nous supposons que le SDI ne contient qu'une unique zone. Une zone a une référence vers un unique *provider* d'informations qui est capable d'agréger les informations à partir des *sources*. Le provider commence donc par interroger le *service d'administration* afin d'obtenir les paramètres d'appels des différentes sources (par exemple, le nom d'un compte Twitter à partir

4. http://tomcat.apache.org/
5. http://www.mongodb.org/

duquel récupérer les informations ou l'URL d'un flux RSS). Une fois les paramètres obtenus, le provider récupère de manière asynchrone les informations à partir de différentes *sources*. Le provider renvoie en réponse à la zone une agrégation des informations obtenues.

Notons que les requêtes asynchrones visent à réduire le temps total de réponse du provider en considérant les différents appels en parallèle : les services étant indépendants de notre système, nous ne pouvons pas garantir leur fiabilité. De même, les différentes zones fonctionnent toutes de manière indépendante afin que le système global ne soit pas bloqué par l'erreur d'une unique zone.

La zone utilise ensuite ces informations en appelant pour chaque type d'information une méthode de transformation de l'information à partir d'un composant de rendu spécifique (ou *renderer*), qui va traiter cette information afin de retourner du HTML qui pourra être affiché. Une fois que toutes les informations ont été traitées dans les différents *renderers*, la zone contient une liste d'informations en HTML qui peuvent être affichées.

Finalement, la zone utilise pour chaque information en HTML une méthode d'affichage à partir du *behaviour* : celui-ci effectue la transition demandée entre les informations.

8.3.2 SDI YourCast et système-de-systèmes

Nous considérons que si le SDI défini dans le projet YOURCAST ne possède pas toutes les caractéristiques d'un système-de-systèmes, il possède en revanche différentes propriétés attendues de variabilité des systèmes-de-systèmes que nous avons identifiées.

Nous ne nous intéressons pas dans nos travaux aux problèmes de passage à l'échelle ou de traitements répartis propres aux système-de-systèmes, mais à la problématique de la modélisation d'une LPL pour un système-de-systèmes. Or les SDI de YOURCAST présentent de nombreux éléments de variabilité que l'on peut considérer comme des systèmes distincts : les différentes *sources* d'informations, par exemple, sont autant de systèmes indépendants que l'on peut utiliser de diverses manières. Par ailleurs, un SDI est créé par la mise en relation des différents systèmes de telle façon qu'un comportement émerge. On retrouve bien les problématiques de représentation et de manipulation de la variabilité à différents niveaux, qui sont les propriétés des systèmes-de-systèmes que l'on cherche à manipuler dans nos travaux.

8.3.3 Modèle du domaine

Nous avons identifié cinq concepts présentant de la variabilité dans le SDI :

— le *layout* est une partie du client gérant à la fois le design de l'écran de diffusion et la disposition des zones, un écran ne peut être basé que sur un seul layout ;
— les *zones* représentent chaque zone de l'écran capable de diffuser indépendamment de l'information : chaque zone spécifie le type de contenu qu'elle peut diffuser (photo, vidéo, texte, etc) et les sous-éléments qu'elle peut contenir (titre, logo, etc) ; une zone est également adaptée pour un design ;
— les *renderers* sont des composants capables d'effectuer le rendu d'une information à partir d'un format générique, adapté pour chaque type d'information : ce sont les renderers qui sont responsables de l'affichage d'un album sous forme d'une mosaïque ou sous forme d'une diapositive ;
— les *behaviours* sont responsables des transitions entre les informations ;
— et enfin les *sources* correspondent aux différentes fonctions d'un service auxquelles on veut pouvoir accéder pour obtenir de l'information : Twitter est un service, récupérer les

tweets issus d'une requête n'est qu'une fonction de l'API Twitter ; il s'agit également d'une *source* pour le SDI.

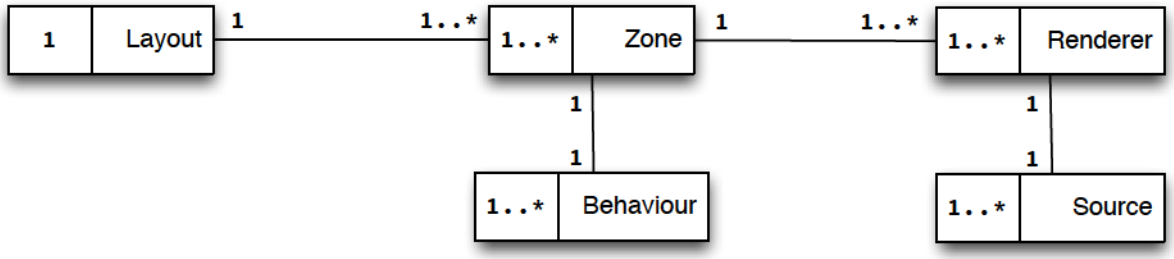


FIGURE 8.4 – Modèle du domaine pour YourCast

La Figure 8.4 représente le modèle du domaine réalisé à partir de ces cinq concepts. Un SDI dans YourCast est donc représenté comme un unique *layout*. Ce layout va pouvoir disposer d'une ou plusieurs *zones* : chaque zone dispose d'un *behaviour* et d'un ou plusieurs *renderers*. Enfin, ces *renderers* sont eux mêmes liés à une unique *source*, chaque *source* disposant d'un unique *renderer* de manière symétrique.

Notons que cette représentation du domaine ne correspond pas à l'architecture logicielle d'un SDI YOURCAST, mais bien à une vision métier du SDI.

8.3.4 Variabilité, FM et Contraintes

Chacun des concepts identifié dans le domaine supporte sa propre variabilité. Nous avons identifié les éléments de variabilités suivants pour les différents concepts :

- layout : variabilité sur le design et les zones contenues ;
- zone : variabilité sur le design, sur les éléments inclus dans la zone et le type de contenu à afficher ;
- renderer : variabilité sur les types d'informations, sur les types de contenus à afficher et sur le design ;
- behaviour : variabilité sur les types d'animation, sur le nombre d'informations à traiter et sur les temps de diffusion ;
- source : variabilité sur les types d'information, sur les services, sur les fonctions pour chaque service et sur les critères de filtrage des informations.

Cependant dans le développement de la LPL YOURCAST, nous n'avons pas souhaité représenter d'emblée l'ensemble de la variabilité des différents concepts. Nous avons en effet préféré suivre une approche itérative de développement de la LPL en ne considérant que les éléments de variabilité dont nous avions besoin au fur et à mesure. Ainsi, afin de rendre possible cette évolution incrémentale de la variabilité, nous avons défini nos FM sous la forme d'une bibliothèque de produits.

8.3.4.1 FM et bibliothèque de produits

La différence entre un FM classique et une bibliothèque de produits tient dans la manière dont ils sont construits. Un FM classique est construit en prévoyant les fonctionnalités des produits inclus dans le scoping. Une bibliothèque de produits correspond plus à une approche bottom-up : le FM est construit directement à partir des produits existants.

L'avantage d'une approche par bibliothèque de produits est qu'elle assure que les produits existants seront représentés et accessibles à travers le FM, mais aussi qu'aucun produit non

disponible ne sera représenté dans le FM. Nous utilisons dans le cadre de YOURCAST le moteur de raisonnement sur les FM FAMILIAR [6] qui définit notamment un opérateur permettant de fusionner des FM tout en conservant leur sémantique [Acher 10b]. L'opérateur garantit en effet que le FM résultant de la fusion représentera un ensemble de configurations correspondant à l'union des ensembles de configurations des FM fusionnés. Ainsi, si on considère qu'à chaque produit correspond une unique configuration, représentée par un FM dans lequel toutes les features sont obligatoires, et que l'on réalise la fusion des FM de tous nos produits, l'opérateur *merge* nous garantit que nous obtiendrons un FM dans lequel nous ne pourrons configurer que les produits d'origine.

Listing 8.1 – Extrait de la représentation de la variabilité des Sources

```
1 source0 = FM(Source : TypeInfo Criteria Provider; TypeInfo : PictureAlbum; PictureAlbum : Picasa;
      Criteria : Filter Limit Sort; Filter : Search; Provider : I3S;)
2 source1 = FM(Source : TypeInfo Criteria Provider; TypeInfo : Events; Events : Supralink_Ateliers;
      Criteria : Filter Limit; Filter : Theme; Provider : Supralog;)
3 source2 = FM(Source : TypeInfo Criteria Provider; TypeInfo : Calendar; Calendar : HyperPlanning;
     Provider : I3S;  Criteria : Filter; Filter : Date CalendarFilter; Date : Beginning;
     CalendarFilter : Room; Room : Occupied;)
4 ...
5 ...
6 source66 = FM(Source : TypeInfo Criteria Provider; TypeInfo : Tweets; Tweets : Twitter; Criteria :
      Filter Limit; Filter : Search; Provider : I3S;)
7 source67 = FM(Source : TypeInfo Criteria Provider; TypeInfo : PictureAlbum; PictureAlbum : Picasa
     ; Criteria : Filter; Filter : Album User; Provider : I3S;)
8
9 yourcast_source = merge sunion source*
```

Nous avons donc réalisé pour chaque concept une bibliothèque de produits. Le Listing 8.1 montre ainsi un extrait du fichier Familiar représentant la variabilité des sources. Ce fichier fait au total 70 lignes et permet de créer un FM représentant 68 produits différents. Chaque ligne représente un unique produit sous la forme d'un FM. Ces différents FM sont écrits dans le langage de représentation des FM de FAMILIAR. Dans ce langage, la hiérarchie est représentée en écrivant "parent : enfant1 enfant2 enfant3". L'absence de tous crochets ou parenthèses dans la syntaxe (excepté pour encadrer la formule du FM) signifie ici que toutes les features sont obligatoires. Par ailleurs, on constate que les FM ont tous une structure extrêmement similaire avec un premier niveau sous la racine contenant à chaque fois les features *TypeInfo*, *Criteria* et *Provider*. En effet, si l'opérateur *merge* garantit de conserver la sémantique des FM en terme de configuration, il ne garantit pas de conserver leur hiérarchie. Ainsi, plus les FM ont une hiérarchie similaire, plus il y a de chance que le résultat de la fusion donne une hiérarchie cohérente pour l'utilisateur final.

Concept	# Features	# Contraintes	# Configuration
Sources	81	154	68
Renderers	76	347	74
Behaviours	33	45	15
Zones	49	160	27
Layouts	51	59	13
Moyenne	58	149	39
Total	290	765	-

TABLE 8.1 – Métriques sur les FM dans YourCast

Le Tableau 8.1 nous donne les métriques obtenues sur les FM finaux obtenus après l'opération

6. http://familiar-project.github.io/

de fusion. On peut constater que si, en moyenne, le nombre de features par FM n'est pas extravagant, le nombre de contraintes est très important en proportion. Cela est dû à l'opération de fusion qui, pour garantir la consistance sémantique, ajoute artificiellement de nombreuses contraintes sur les FM.

La Figure 8.5 donne ainsi un aperçu de la structure complexe du FM obtenu pour représenter la variabilité des *sources* d'informations. La partie droite de la figure a été coupée : il s'agit de l'ensemble des contraintes du FM représentées de manière textuelle.

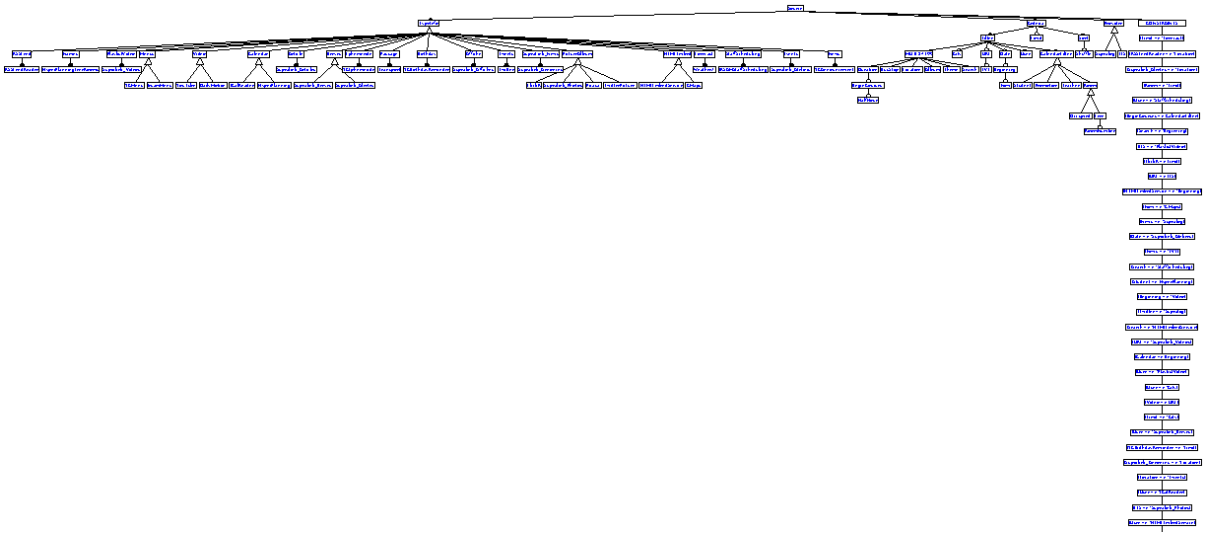


FIGURE 8.5 – Variabilité des *sources* dans YOURCAST

8.3.4.2 Fonctions et règles de restrictions

Les fonctions de restriction représentent les facteurs de compatibilité entre les produits des différents concepts. La compatibilité est représentée de la manière suivante sur les différentes associations :

- entre les sources et les renderers : la compatibilité est définie par le type d'information renvoyé par la source et consommé par le renderer ;
- entre les renderers et les zones : la compatibilité est définie par le type de contenu renvoyé par le renderer et supporté par la zone ;
- entre les behaviours et les zones : la compatibilité est définie entre le type de contenu supporté par la zone et manipulable par le behaviour ;
- entre les zones et les layouts : la compatibilité est définie par le respect d'un même design et le support des types de zones par le layout.

Listing 8.2 – Fichier de définition d'une fonction de restriction sur l'association zone-layout

```
1 source=Layout
2 target=Zone
3 rule :
4         SELECTED Design.$A => SELECT Design.$A ;
5
6 rule :
7         DESELECTED Zones.$A => DESELECT Product.$A ;
```

Le Listing 8.2 donne un exemple de définition de fonction de restriction écrit dans notre DSL dédié (voir section 7.4). La fonction définie ici est orientée de Layout vers Zone. Elle définit deux catégories de règles et non deux règles comme on pourrait le croire. La ligne 4 va

en effet créer autant de règles qu'il y a d'enfants à la feature *Design* du FM de *layout* : elle va chercher à établir une relation avec les enfants de la feature *Design* cette fois du FM *zone*, qui ont le même nom. La ligne 7 réalise la même opération mais pour les enfants de la feature *Zones* du FM *layout* et les enfants de la feature *Product* du FM *zone*. Au total de nombreuses règles sont créées ainsi de manière automatique, les règles inverses étant ensuite également calculées automatiquement comme défini dans la sous-section 5.4.5.

Association	# Fonctions de restriction	# Règles de restriction
Source - Renderer	2	40
Renderer - Zone	2	20
Behaviour - Zone	2	10
Zone - Layout	2	74
Moyenne	2	36
Total	8	144

TABLE 8.2 – Métriques sur les règles et fonctions de restriction dans YourCast

Le Tableau 8.2 nous montre la répartition des fonctions et règles de restriction entre les différentes associations. On constate en premier lieu que chaque association a exactement deux fonctions de restriction : nous avons défini, en réalité, pour chaque association une seule fonction de restriction, la seconde est la fonction contraposée qui est calculée automatiquement. Dans l'absolu, le nombre de fonctions de restriction est toujours pair. Le nombre de règles varie en revanche beaucoup en fonction des associations, nous constatons cependant qu'il est également systématiquement pair. Cela est vrai pour nous parce que nos règles sont toujours simples dans YourCast et ne sont constituées que d'un état avec une unique feature : dès lors, la contraposée donne lieu à une unique règle également. Cependant, comme expliqué dans la sous-section 5.4.5 cela n'est pas toujours le cas pour des règles plus complexes où la contraposée peut donner lieu à de multiples règles. Par ailleurs, le nombre variant de règles est généralement corrélé à la taille des FM : le FM behaviour est le plus petit dans le Tableau 8.1 et on voit que l'association behaviour-zone a le plus petit nombre de règles. Cela est dû au fait, comme présenté dans le Listing 8.2, que les règles sont calculées automatiquement à partir des enfants d'une feature, établissant une corrélation directe entre le nombre de sous-features et le nombre de règles.

8.4 Objectifs et résultats

Nous avons montré dans la section précédente les choix réalisés afin de modéliser la variabilité d'une LPL permettant de produire des systèmes de diffusion d'information. Nous cherchons à valider ici le premier objectif de notre thèse qui concernait la modélisation de la variabilité des LPL complexes garantissant une bonne séparation des préoccupations et des formalismes, ainsi que la cohérence des éléments de variabilité.

Nous revenons dans cette section sur les différents sous-objectifs relatifs à ce point et comment ils ont été atteints dans le cadre de la LPL YOURCAST.

8.4.1 Représentation d'une LPL par l'utilisation d'un modèle du domaine et de modèles de variabilité

Une première validation de cet objectif est donnée directement par notre représentation de la variabilité (voir chapitre 5). La variabilité est en effet définie à la fois par un modèle du domaine

présentant un haut niveau d'abstraction et par l'exploitation de différents modèles de variabilité. Par ailleurs, le modèle du domaine permet d'exprimer des multiplicités en terme d'intervalle, aussi bien sur les concepts de la LPL, que sur les associations entre ces concepts.

Nous validons cette représentation d'une LPL dans le cadre du projet YOURCAST. En effet, cette approche nous a permis de séparer la variabilité d'une LPL dédiée à un SDI au sein de multiples FM ayant chacun une responsabilité clairement identifiée. Par ailleurs, le modèle du domaine de la LPL YOURCAST ne définit que 5 concepts et se comprend très rapidement : il permet d'avoir une représentation abstraite d'un produit de SDI, alors même que la variabilité des SDI est très complexe.

Nous montrons ainsi dans le Tableau 8.1 que nous totalisons dans la LPL YOURCAST environ 300 features pour 765 contraintes internes et 144 règles de restrictions entre les FM (voir Tableau 8.2). Représenter cette variabilité au sein d'un unique FM le rendrait très difficilement compréhensible et utilisable, là où des FM contenant une soixantaine de features en moyenne sont beaucoup plus faciles à gérer. La Figure 8.5 montre déjà la complexité d'un FM de cette taille, ce qui nous conforte dans le besoin de maîtriser une bonne séparation des préoccupations dans la représentation de la variabilité.

8.4.2 Garantir la cohérence de la LPL

La cohérence de la LPL est garantie par les différentes propriétés formalisées dans notre contribution (voir section 5.6) et par l'algorithme que nous avons défini pour vérifier la cohérence la LPL (voir section 6.8). Nous précisons ces points ci-après.

8.4.2.1 Cohérence du modèle du domaine

Dans le cadre de la LPL YOURCAST, les propriétés de cohérence du modèle du domaine (voir sous-section 5.2.5) peuvent être vérifiées immédiatement à la seule vue de la Figure 8.4. En effet ce modèle est connexe : chaque concept peut être accessible à partir de n'importe quel autre. De plus il ne présente aucune relation réflexive et les multiplicités exprimées sont toutes cohérentes entre elles.

8.4.2.2 Cohérence des FM

Dans le cadre de la LPL YOURCAST, la validation des propriétés de cohérence des FM (voir sous-section 5.6.2) est inutile, les FM étant construits comme des bibliothèques de produits. Cette méthode nous assure que chacune des features des différents FM appartiendra forcément à une configuration valide du FM. Il n'existe alors pas de feature morte et il existe au minimum toujours un produit valide du FM.

Cependant, la validation des FM peut être réalisée de manière indépendante en utilisant les opérations d'analyse permises par FAMILIAR.

8.4.2.3 Cohérence des règles de restriction

Si, comme nous l'avons discuté dans la sous-section 5.6.3, le calcul de cohérence des fonctions de restriction possède une complexité bornée, il n'en reste pas moins difficile. Nous avons dans le cadre de nos travaux, uniquement implémenté un algorithme de détection naïf des contradictions entre les règles de restrictions : si une action de sélection ou d'exclusion est en contradiction avec une action précédemment exécutée, alors une exception est immédiatement levée pour informer l'utilisateur que la LPL n'est pas cohérente.

Par ailleurs, nous considérons que notre test de réalisabilité nous permet d'assurer en partie la cohérence des règles de restriction relativement aux sous-configurations autorisées dans la LPL.

8.4.2.4 Vérification de la réalisabilité

La vérification de la réalisabilité doit s'effectuer en exploitant notre algorithme de réalisabilité (section 6.8). Nous constatons que la topologie du modèle de domaine de YOURCAST se prête bien à l'algorithme que nous avons défini. En effet, le modèle présenté dans la Figure 8.4 possède de nombreuses composantes biconnexes. Les nœuds *renderer* et *zone* sont tous deux des points d'articulation, les composantes biconnexes sont donc :

— layout-zone,
— zone-behaviour,
— zone-renderer,
— renderer-source.

Ainsi pour tester si un renderer peut appartenir à une configuration composite, il suffit de vérifier qu'il existe une solution dans ses deux composantes biconnexes (zone-renderer et renderer-source), soit pour seulement deux concepts de la LPL.

Nous ne validons ce point que sur sa partie théorique, l'implémentation de l'algorithme n'ayant pas été achevée lors de l'écriture de ce document.

8.4.3 Exploiter des standards et le point de vue de l'utilisateur final pour la définition de la LPL

Nous validons cet objectif en considérant indépendamment ses deux aspects : l'exploitation des standards, et la prise en compte du point de vue utilisateur pour la définition de la LPL.

8.4.3.1 Utiliser des standards

Nous validons tout d'abord cet objectif par le formalisme que nous proposons pour la représentation de la variabilité. En effet, nous exploitons dans notre contribution une formalisation des FM, qui sont les modèles les plus utilisés pour représenter la variabilité d'après [Berger 13]. Par ailleurs, nous avons défini notre formalisation des FM pour la rendre la plus compatible possible avec les différentes représentations existantes de FM : elle ne contient que le minimum des concepts nécessaire à la représentation d'un FM.

De même, notre formalisation du modèle du domaine ne comporte que le minimum de concepts nécessaires à représenter un modèle entité/relations possédant des multiplicités. De fait, il est facile de réaliser des transformations de modèles à partir de différents formalismes de modèles objets pour obtenir un modèle qui soit conforme à notre formalisation.

En outre, l'implémentation même de notre formalisme et les outils que nous avons employés convergent vers l'utilisation de standard. Le métamodèle de SPINEFM, qui implémente l'ensemble de notre formalisme (voir section 7.3), a été réalisé en utilisant l'outil de modélisation EMF défini par Eclipse et qui est directement dérivé du MOF, considéré comme un standard pour la métamodélisation.

Par ailleurs, si FAMILIAR n'est pas un outil standard pour la définition et l'exploitation de la variabilité, il a l'avantage d'être capable d'interpréter de nombreux formalismes de FM. Il est ainsi compatible avec les formalismes proposés par FeatureIDE, SPLOT, TVL, etc. L'utilisation

de FAMILIAR dans notre implémentation fait ainsi du sens dans cette volonté d'atteindre le plus grand nombre de formalismes de FM possibles.

8.4.3.2 Définir la variabilité selon le point de vue de l'utilisateur final

Comme nous l'avons expliqué au début de ce chapitre, le modèle du domaine de la LPL YOURCAST a été défini en reprenant les différents concepts métier des SDI. Nous avons par ailleurs, dans la mesure du possible, défini la variabilité en continuant à exploiter un vocabulaire métier lié aux différentes préoccupations des FM, ainsi que des hiérarchies qui font du sens du point de vue métier [Blay-Fornarino 12]. La variabilité des *sources* est ainsi bien définie selon les deux grands axes que sont le type d'information et la manière de filtrer cette information.

Par ailleurs, nous exploitons directement dans l'interface graphique de configuration TOC-SIN les informations du domaine, ce qui justifie d'autant plus le besoin d'exprimer la variabilité selon le point de vue de l'utilisateur final. Une analyse ergonomique réalisée par une société indépendante sur nos interfaces de configuration dans le cadre du projet YOURCAST a ainsi montré que les utilisateurs comprenaient bien les différents concepts de la LPL et les liens qui existaient entre eux. Cependant, l'expression de la variabilité au sein des FM dans toute sa finesse reste un exercice difficile, ce qui justifie l'exploitation des métadonnées au sein des interfaces de configuration afin d'apporter des informations supplémentaires sur les différentes features comme des explications ou des logos afin de faciliter l'identification des features.

8.5 Conclusion

Nous validons ainsi dans ce chapitre notre approche de modélisation de la variabilité sur une LPL de portée industrielle destinée à la création de systèmes de diffusion d'information. Nous avons montré dans quelle mesure ce cas d'utilisation correspondait à une LPL complexe, offrant une grande variabilité, notamment dans la composition de ces différents éléments. Nous avons ensuite discuté la validation de nos différents objectifs en relation avec les différentes propriétés définies dans notre contribution, et les résultats obtenus sur la LPL que nous avons définie.

CHAPITRE 9

RÉALISER DES PRODUITS DANS LA LPL YOURCAST

Sommaire

Résumé Nous validons dans ce chapitre la seconde partie de notre approche relative à la cohérence et la flexibilité du processus de configuration par l'exploitation de la LPL YOURCAST précédemment décrite.

Nous décrivons ainsi dans ce chapitre non seulement les configurations réalisées au sein de la LPL YOURCAST, mais également le processus de réalisation utilisé dans cette LPL. Enfin, nous validons les objectifs que nous avions explicités au début de ce manuscrit par l'analyse des résultats obtenus sur une dizaine de configurations distinctes.

9.1 Introduction

La LPL YourCast a été utilisée pour effectuer des déploiements en production de SDI qui sont encore utilisés en 2015. Si les déploiements des SDI n'ont pas été automatisés, la LPL nous a permis en revanche de générer intégralement les différents éléments des SDI à déployer suite aux configurations composites réalisées. Nous revenons ici sur le processus de dérivation global d'un SDI dans la LPL YourCast en définissant précisément notre processus de génération, avant de discuter nos différents éléments de validation.

9.2 Processus de configuration

L'interface graphique de configuration TOCSIN (voir section 7.6) a été utilisée pour réaliser l'ensemble des configurations composites de la LPL YourCast.

Des métadonnées ont été créées pour l'ensemble des concepts afin de définir les catégories d'affichage pour la configuration des différents concepts et afin de déterminer les explications des différentes features. En outre, des images ont été réalisées afin de représenter graphiquement la grande majorité des features des FM pour aider l'utilisateur dans ses choix.

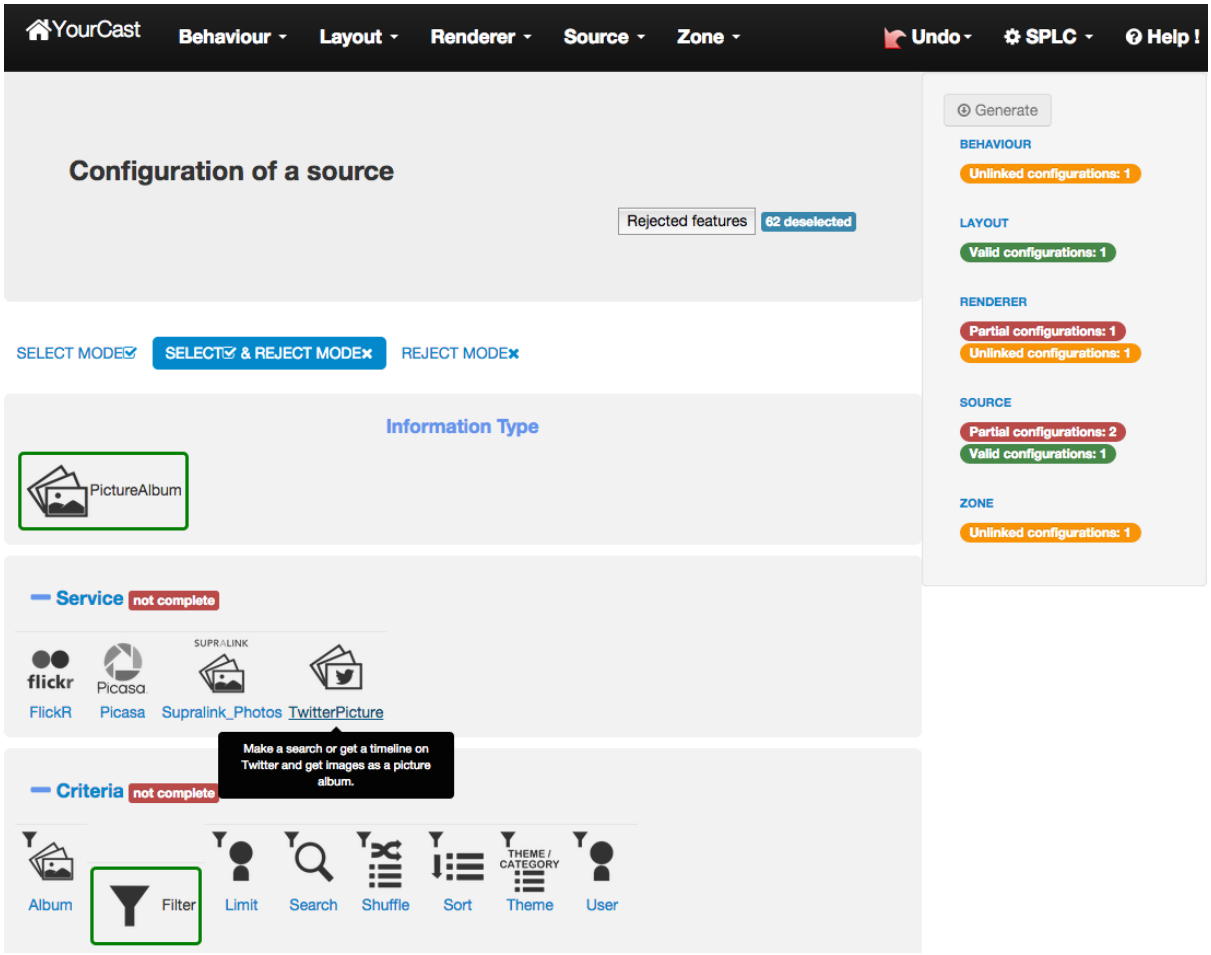


FIGURE 9.1 – Capture d'écran de TOCSIN pour la LPL YourCast

La Figure 9.1 montre ainsi une capture d'écran de l'interface de configuration TOCSIN utilisée dans le cadre de la LPL YourCast. Cette capture a été effectuée dans le cadre de

la réalisation d'une sous-configuration de *Source*, on peut voir ici les différentes catégories d'affichage des features pour ce concept, ainsi qu'une explication pour la feature *TwitterPicture*.

Par ailleurs la configuration composite n'est pas terminée : on peut voir en haut de la colonne de droite que le bouton de génération (*Generate* dans l'image) est toujours grisé. Des informations supplémentaires sont disponibles dans la colonne de droite indiquant que des sous-configurations de différents concepts ne sont pas encore liées (texte avec un fond orange) ou ne sont pas encore terminées (texte avec un fond rouge).

9.3 Processus de génération

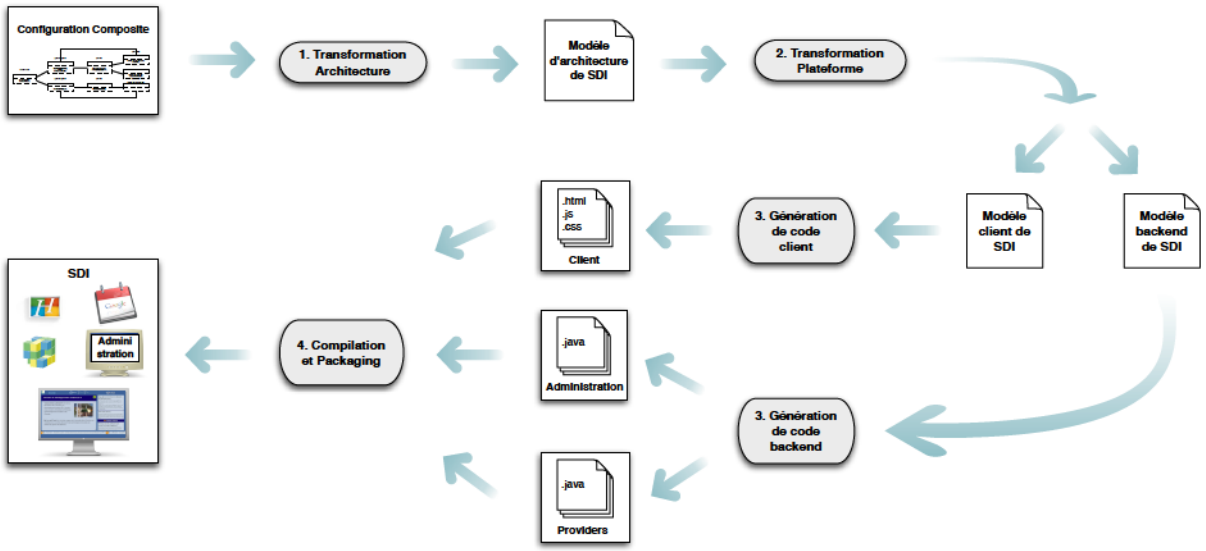


FIGURE 9.2 – Vue d'ensemble du processus de génération dans la LPL YOURCAST

La Figure 9.2 donne une vue d'ensemble du processus de génération dans le cadre de la LPL YOURCAST. Cette vue d'ensemble montre que le processus de génération se déroule en 4 étapes principales que nous détaillons par la suite. Le processus a été défini selon les principes de l'ingénierie dirigée par les modèles : il utilise différentes transformations pour, à partir d'une représentation très abstraite d'un système, aboutir à une implémentation concrète compilée et packagée du système représenté.

Toutes les opérations du processus de génération ont été écrites en Java en exploitant notamment les possibilités offertes par le framework Eclipse EMF, déjà exploité pour SPINEFM (voir chapitre 7). Ce travail a été réalisé de manière collaborative dans le cadre du projet YOURCAST, nous avons notamment participé à la définition des différents métamodèles intermédiaires.

9.3.1 Transformation vers un modèle d'architecture

La première étape du processus de génération de YOURCAST consiste à produire un modèle d'architecture à partir de la configuration composite réalisée durant le processus de configuration. Un métamodèle d'architecture a été défini permettant de représenter les différents concepts d'un SDI et la manière dont ils sont interconnectés. Une représentation simplifiée en est donnée dans la Figure 9.3.

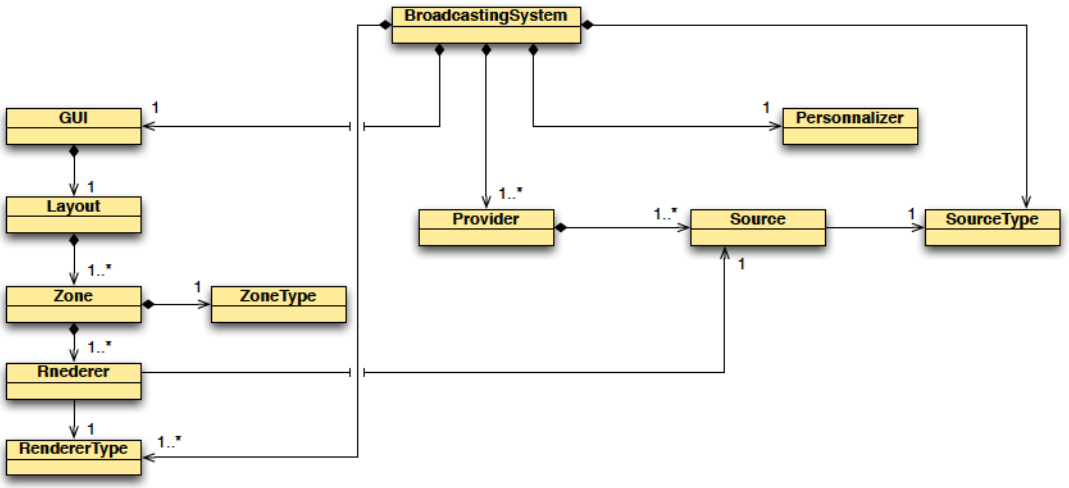


FIGURE 9.3 – Représentation simplifiée du métamodèle d'architecture de YOURCAST

Le modèle d'architecture créé automatiquement est conforme au métamodèle défini et représente l'architecture du SDI configuré. Par ailleurs, ce modèle contient des références vers des artefacts logiciels en lieu et place des features.

Pour cela, l'opération de transformation a besoin d'identifier pour chaque sous-configuration les différents artefacts logiciels qu'elle représente. Nous avons défini pour chaque concept de YOURCAST un métamodèle explicitant les types d'artefact logiciel qu'une configuration de FM doit représenter, nous l'appelons un *métamodèle d'asset*.

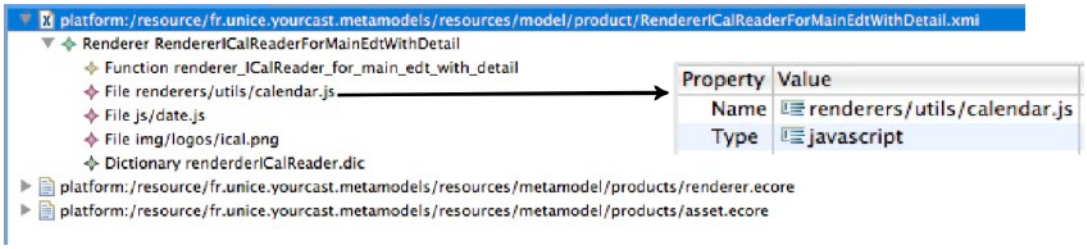


FIGURE 9.4 – Exemple d'un modèle d'asset de *Renderer* pour YOURCAST

La Figure 9.4 donne ainsi un exemple de modèle d'asset pour un *Renderer* particulier. Cette figure montre qu'un *renderer* référence différents artefacts tels qu'un fichier javascript, une image, etc.

Nous avons défini un outil permettant de décrire un modèle conforme au métamodèle de chaque concept et lui associer une représentation de FM dans la syntaxe de FAMILIAR. Ainsi cet outil nous permet à la fois de décrire les FM de chaque concept de manière incrémentale, mais également de réaliser automatiquement le mapping entre une configuration de FM et un modèle d'asset.

L'opération de transformation d'une configuration composite vers un modèle d'architecture commence par itérer sur l'ensemble des sous-configurations de la configuration composite pour obtenir chacun des modèles d'assets auxquelles elles sont associées : elle exploite pour cela le modèle du domaine de la LPL afin d'obtenir le nom des différents concepts référencés par les sous-configurations.

9.3.2 Transformation vers des modèles de plateforme

La deuxième transformation a pour objectif de passer d'une représentation architecturale du système à une représentation abstraite tenant compte des choix technologiques effectués pour la plateforme du produit. Dans notre cas, les choix technologiques sont très divergents entre la partie *cliente* du système, réalisée grâce à des technologies web comme HTML et javascript ; et la partie arrière du système que l'on qualifie de *backend* : cette partie du système est essentiellement réalisée en utilisant le langage Java pour être déployé sur un serveur d'application Tomcat.

Nous avons donc défini deux métamodèles distincts, l'un afin de représenter de manière spécifique un *client* de YOURCAST et l'autre afin de représenter un *backend* contenant des *sources*, *providers*, et une *administration*. L'opération de transformation a donc pour tâche de créer les deux modèles conformes aux métamodèles de plateforme, à partir du modèle d'architecture précédemment réalisé.

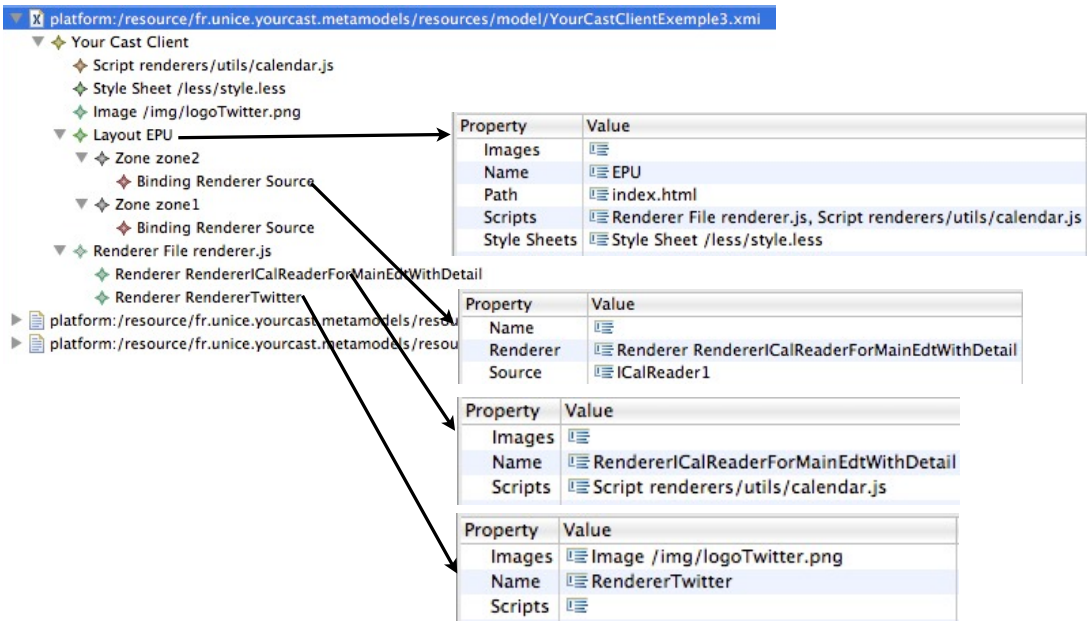


FIGURE 9.5 – Exemple d'un modèle de plateforme pour un *Client* de YOURCAST

La Figure 9.5 montre un exemple de modèle de plateforme pour le *client* d'un SDI configuré dans la LPL YOURCAST. On constate la présence d'informations qui proviennent directement des modèles d'assets précédemment évoqués.

9.3.3 Génération de code

La troisième étape du processus de génération consiste à réaliser la génération des codes concrets du SDI configuré. Pour cela, les modèles de plateformes réalisés à l'étape précédente sont exploités afin de créer ou réutiliser les différents codes qui leur correspondent et les assembler dans une hiérarchie de fichiers prédéfinie.

Cette opération se déroule donc en deux étapes pour chacun des types de plateforme. La première étape consiste à récupérer l'ensemble des fichiers auxquels fait référence le modèle de plateforme. La seconde étape consiste à générer du code dans certains fichiers prédéfinis à partir des informations contenues dans le modèle de plateforme. Nous utilisons dans le cadre de la

LPL YOURCAST le moteur de template Apache Velocity [1] afin de faciliter la génération de code.

L'étape de génération de code produit ainsi dans le cadre de YOURCAST plusieurs projets distincts contenant leur propre hiérarchie de code : un projet représentant le client, un autre représentant le service d'administration et enfin différents projets pour chacun des providers nécessaires.

Element	Code commun	Code variant [2]	Total
Client	10545	4760	15305
Administration	58915	32403	91318
Provider 1	839	288	1127
Provider 2	839	232	1071
Provider 3	839	250	1089
Provider 4	839	231	1070
Total	72816	38164	110980

TABLE 9.1 – Métriques de code pour un SDI moyen

Le Tableau 9.1 donne quelques chiffres concernant la densité de codes générés ou réutilisés dans le cadre d'un SDI moyen contenant 4 zones, et donc 4 providers. Toutes les valeurs données sont en lignes de code. La majorité du code généré est réalisée au sein du système d'administration afin de créer les différentes interfaces de personnalisation en fonction des sources sélectionnées par l'utilisateur et des liens réalisés. Enfin, une partie importante du client est également générée ou réutilisée en fonction des choix utilisateurs. Ces mesures correspondent à la configuration A du tableau présenté plus loin.

9.3.4 Compilation et packaging

La dernière étape du processus de génération réalise la compilation et le packaging des différents projets de code réalisés à l'étape précédente. Ainsi les différents projets générés dans le langage Java sont compilés et packagés pour être déployés sur un serveur d'application de type Tomcat. Dans le cadre de YOURCAST nous exploitons les facilités offertes par l'outil de gestion et d'automatisation de production de logiciels Apache Maven [3] afin de réaliser ces opérations.

Le client est par ailleurs compressé sous la forme d'une archive déployable également sur un serveur d'application.

Ainsi le processus de génération part d'une configuration composite et exploite différents métamodèles en association avec un dépôt d'assets afin de réaliser un logiciel compilé et packagé, prêt à être déployé.

9.4 Objectifs et Résultats

Nous avons défini et utilisé la LPL YOURCAST en exploitant l'approche définie dans notre contribution (voir chapitres 5 et 6).

Nos travaux sur les processus de configuration visent à définir un processus qui soit à la fois flexible et cohérent, mais qui soit également utilisable (voir chapitre 4). Nous présentons à

1. http://velocity.apache.org/
2. Il peut s'agir d'un code généré en partie ou totalement, ou simplement réutilisé.
3. http://maven.apache.org/

présent les différents processus de configurations opérés et les résultats obtenus, avant de revenir sur nos deux objectifs et leurs validations.

9.4.1 Configurations et résultats obtenus

La LPL YOURCAST a été utilisée pour réaliser différents types de SDI en fonction des déploiements qui étaient demandés dans le cadre du projet YOURCAST. Nous nous concentrons ici uniquement sur dix configurations de SDI dont des mesures sont présentées dans le Tableau 9.2. De nombreux SDI ont été configurés et générés à partir de cette LPL (plus d'une vingtaine de SDI différents ont été déployés durant le projet YOURCAST), nous nous focalisons cependant ici sur un échantillon représentatif de SDI réalisés dans différentes conditions.

	A	B	C	D	E	F	G	H	I	J	Moy.
Sous-Config.	37	9	25	17	23	61	11	12	7	21	22,3
Liens	36	8	24	16	22	60	8	9	6	20	20,9
Contextes	19	5	13	9	12	32	9	8	7	11	12,5
CPS	77	21	53	37	49	129	37	33	29	45	51
Actions Util.	259	68	194	147	172	348	78	125	51	138	158
Actions Sys.	4111	849	2835	1727	2703	7218	1416	1587	942	1965	2535,3
Actions Auto.	631	140	462	241	399	1492	299	239	178	213	429,4
Actions FM.	3480	709	2373	1486	2304	5726	1117	1348	764	1752	2105,9
% Actions Util.	5,93	7,42	6,40	7,84	5,98	4,60	5,22	7,30	5,14	6,56	5,87
% Actions Sys.	94,07	92,58	93,60	92,16	94,02	95,40	94,78	92,70	94,86	93,44	94,13

TABLE 9.2 – Résultats obtenus suite aux configurations YourCast

Les six premières colonnes de ce tableau (de A à F) ont ainsi été obtenues lors de configurations réalisées par des experts de la LPL, soit pour déploiements réels (de A à D), soit pour tester des propriétés de la LPL (E et F). Les quatre colonnes suivantes (G à J) concernent des résultats obtenus dans le cadre d'expérimentations sur l'ergonomie de nos interfaces par des personnes qui n'étaient pas expertes dans l'utilisation des outils de configuration, ni même dans les SDI (personnels administratifs ou de communication). Enfin la dernière colonne présente les moyennes obtenues.

Les deux premières lignes de résultats concernent le nombre de sous-configurations et de liens obtenus dans la configuration composite. Les deux lignes suivantes présentent le nombre total de contextes et de CPS utilisés lors de la configuration, les contextes et CPS supprimés sont donc comptés. Les deux lignes d'après montrent le nombre d'actions utilisateur et système réalisées durant le processus de configuration. Les actions système sont divisées en deux catégories dans les lignes suivantes : les actions portant sur les FM et les autres actions automatisées. Enfin les deux dernières lignes donnent le ratio en pourcentage du nombre d'actions utilisateur et système sur le nombre total d'actions.

Nous avons également agrégé des données sur notre algorithme de propagation durant ces processus de configuration.

Chemin	# Occurences	Durée (ms)	# Actions	# Contextes impactés [4]	# CPS impactés
1	141	1,81	9,77	1	1
2	116	2,23	25,02	1,16	1,83
3	60	584,18	32,55	1,05	2,9
4	7	9,29	32,43	1,57	3,14
5	30	99	29,77	2 [CG]	5
6	35	730,26	51,40	1,91 [CG]	4,91
7	36	1607,08	47,97	1,92 [CG]	5
9	1	2109	78	2 [CG]	5
10	2	38	57	2,5 [CG]	7
13	1	1575	52	2 [CG]	9
16	2	1393	77	2 [CG]	9
18	2	785	118,5	2,5 [CG]	7
19	1	798	57	2 [CG]	9
20	1	2035	95	2 [CG]	9
26	3	1556,67	137	3 [CG]	13
27	1	1572	81	7 [CG]	21
36	1	3622	218	7 [CG]	29
40	2	878	120	2,5 [CG]	11

TABLE 9.3 – Résultats obtenus sur les propagations

	Chemin	Durée (ms)	# Actions	# Contextes impactés	# CPS impactés
Minimum	1	0	1	1	1
Maximum	40	4206	218	7	29
Moyenne	3,54	327,11	28,54	1,35	2,81
Ecart type	4,55	708,58	24,25	0,62	2,54

TABLE 9.4 – Statistiques globales obtenues sur les propagations

Celles-ci sont présentées dans le Tableau 9.3. Durant les 10 processus de configuration réalisés, nous avons ainsi comptabilisé 442 lancements de l'algorithme de propagation, en dehors des appels de récursivité de l'algorithme. Nous représentons dans le Tableau 9.3 les données agrégées sur ces propagations en les groupant en fonction de la taille des chemins de propagation réalisés. La taille d'un chemin de propagation est déterminée par le nombre d'appels récursifs réalisés. Nous représentons cette donnée dans la première colonne du tableau.

La seconde colonne du tableau représente le nombre d'occurences de chaque taille de chemin de propagation. La troisième colonne donne les durées moyennes de propagation en fonction de la taille du chemin. La quatrième colonne montre le nombre d'actions système réalisées en moyenne. La cinquième colonne présente le nombre moyen de contextes impactés durant la propagation ; nous indiquons également dans cette colonne par la mention "[CG]" si le contexte global a été impacté plus de 50% du temps. Enfin, la sixième colonne donne le nombre moyen de CPS qui ont été impactés par la propagation.

Le Tableau 9.4 reprend les mêmes colonnes que le tableau précédent mais établit cette fois des statistiques globales basées sur les 442 valeurs confondues obtenues durant la propagation.

L'ensemble des mesures relatées ici ont été effectuées sur une machine virtuelle exploitant un système d'exploitation Linux Debian version 6.0.8, possédant 4096 Mo de mémoire vive

4. La mention "[CG]" signifie que le contexte global a également été impacté dans la majorité des cas (> de 50%).

(RAM) et 4 CPU cadencés à 2,4 Ghz. Nos outils étaient lancés en tant que services sur le serveur d'application Apache Tomcat 6 limité à 512 Mo de mémoire. La version utilisée de la machine virtuelle Java pour nos expérimentations était la version 1.6.0 build 26.

Nous discutons l'ensemble de nos résultats au travers de la validation de nos différents objectifs.

9.4.2 Flexibilité et cohérence du processus de configuration

La flexibilité et la cohérence du processus de configuration constitue le deuxième objectif de nos travaux de thèse (voir section 4.3). Nous validons cet objectif en reprenant les différents sous-objectifs qu'il définit.

9.4.2.1 Garantir un processus de configuration dynamique et cohérent

Nous avons effectué une validation théorique de la dynamicité et de la cohérence du processus de configuration à travers notre contribution (voir chapitre 6). En effet, nous avons défini formellement les propriétés de cohérence et de flexibilité du processus de configuration (Propriété 6.1.1) avant de montrer comment notre formalisme associé à nos algorithmes de propagation (section 6.4) et d'annulation (sous-section 6.5.3) permettent d'assurer ces propriétés. Par ailleurs, nous avons également montré comment assurer la dynamicité du processus de configuration en utilisant les algorithmes de satisfiabilité pour assurer l'inférence automatique des contraintes sur les FM (section 6.2).

Les configurations présentées dans le Tableau 9.2 sont toutes des configurations cohérentes selon la Propriété 5.3.3 et ont été obtenues grâce à l'implémentation de notre formalisme qui s'appuie sur des outils permettant d'assurer la dynamicité de la configuration.

9.4.2.2 Permettre un processus de configuration par étapes, ainsi que l'annulation d'étapes et la reprise de configuration

Nous avons défini dans notre formalisme un modèle d'action dans lequel nous distinguons les actions utilisateur et les actions système (section 6.5). Nous considérons par ailleurs un historique de configuration (sous-section 6.5.2) contenant les différentes étapes de configurations définies par les actions utilisateur.

Si nous n'avons pas pu obtenir d'informations concernant l'utilisation des opérations d'annulation durant le processus de configuration, la fonctionnalité ayant été développée et intégrée tardivement à l'interface TOCSIN, nous avons cependant agrégé de nombreuses informations concernant les actions réalisées durant différents processus de configuration (voir Tableau 9.2).

Nous montrons ainsi, qu'en moyenne, si les utilisateurs réalisent 158 actions pour définir un SDI constitué de 22,3 éléments distincts, 2535,3 actions système sont réalisées automatiquement, parmi lesquelles 2105,9 actions sur les FM. Cela revient à dire que notre approche permet d'automatiser plus de 94% des actions afin de configurer un système, sans compter les vérifications effectuées pour chacune des actions.

En outre, nos outils permettent d'enregistrer et d'exporter une configuration sous la forme d'une liste d'actions utilisateur. Cela nous offre la possibilité de réimporter des configurations composites existantes, *même si la LPL a évolué* : en effet, l'importation d'une configuration consiste à exécuter les actions utilisateur ; si la LPL a évolué et qu'une action utilisateur n'est plus possible, alors la précondition de l'action interdit son action, sans empêcher le reste de l'importation de la configuration. Cette possibilité, combinée aux opérations d'annulation, nous

permet de démarrer des configurations en utilisant les anciennes comme des modèles dans lesquels il est possible de changer certaines décisions.

9.4.2.3 Permettre la réalisation d'un processus de configuration cohérent sans ordre prédéfini

La conservation de la cohérence du processus de configuration sans ordre est assurée à la fois par notre algorithme de propagation (section 6.4) et par les préconditions données aux actions.

Nous avons ainsi analysé le processus de configuration des 10 configurations répertoriées dans le Tableau 9.2 en nous intéressant aux différents appels de l'algorithme de propagation.

L'analyse de lancement des différents appels à l'algorithme de propagation nous informe sur le processus de configuration de l'utilisateur. En effet, il nous permet de savoir quel a été le chemin de l'utilisateur de concepts en concepts pour créer son SDI. Nous avons ainsi pu constater que tous les utilisateurs ont eu un workflow de configuration différent pour la configuration de leur SDI.

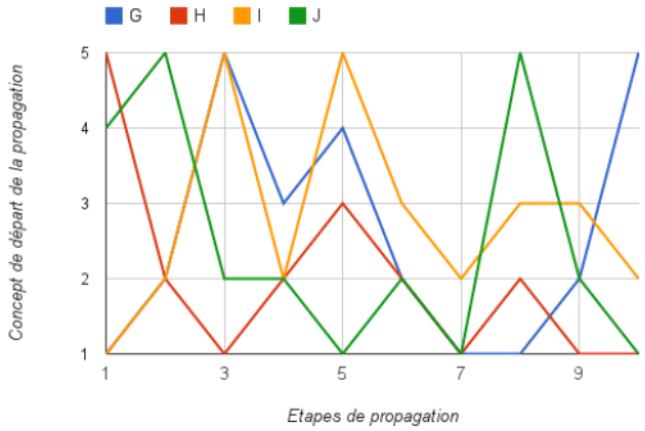


FIGURE 9.6 – Workflow de configurations des utilisateurs non-experts

La Figure 9.6 montre les 10 premières étapes du workflow de configuration pour les utilisateurs non-experts (soit les configurations de G à I). Les nombres en ordonnées correspondent aux différents concepts de la LPL (1 : *source*, 2 : *renderer*, 3 : *behaviour*, 4 : *layout* et 5 : *zone*). Cette figure nous donne ainsi une représentation visuelle des différences entre les workflows utilisateurs.

Nous avons obtenu 442 appels différents à l'algorithme de propagation, en dehors des appels récursifs, dont les résultats sont synthétisés dans le Tableau 9.3. Nous nous sommes intéressés dans ce tableau à analyser les différents appels de l'algorithme de propagation en fonction de la taille des chemins parcourus durant la propagation. En effet, notre algorithme est récursif et bien que nous ayons montré qu'il se terminait toujours en théorie, il nous semble pertinent de montrer expérimentalement qu'en plus de se terminer toujours, les appels récursifs effectués durant l'algorithme sont très limités. Ainsi, le tableau comptabilise des chemins de propagation allant jusqu'à 40 étapes. Cependant, on constate que 425 appels de l'algorithme n'ont fait que des chemins d'une longueur inférieure ou égale à 7 étapes, soit 96% des appels. Cela se trouve confirmé par la valeur moyenne et l'écart type de la taille des chemins montrés dans le Tableau 9.4.

En outre, le tableau montre que tous les appels totalisant plus de 7 étapes de propagations ont permis l'application d'un grand nombre d'actions, supérieur à 55 actions en moyenne et

pouvant aller jusqu'à 218 actions appliquées. Cela montre que si l'algorithme de propagation fait beaucoup de récursion, il réalise en contrepartie un grand nombre d'applications d'actions, ce qui facilite le processus de configuration.

9.4.3 Utilisabilité du processus de configuration

Le dernier objectif de nos travaux de thèse concernait l'utilisabilité du processus de configuration (voir section 4.4). Nous validons dans la suite l'utilisabilité selon deux aspects : le guidage des utilisateurs durant le processus de configuration, et la garantie des temps de réponse et d'une empreinte mémoire raisonnables dans les outils.

9.4.3.1 Permettre le guidage des utilisateurs durant le processus de configuration

Le guidage des utilisateurs est intrinsèquement lié aux interfaces de configurations utilisées durant le processus. Nous avons présenté dans la section 7.6, TOCSIN, l'interface graphique de configuration que nous avons réalisée spécifiquement pour notre approche.

TOCSIN exploite les informations du modèle du domaine et les informations de la configuration en cours de réalisation afin de créer dynamiquement les interfaces en guidant l'utilisateur selon les associations exprimées dans le modèle du domaine. Nous validons en partie cette approche par la réalisation de la part de différents utilisateurs de plusieurs configurations composites dans lesquelles les configurations sont effectivement liées. Nous pouvons ainsi constater sur le Tableau 9.2 qu'en moyenne 21 liens sont créés pour 22 sous-configurations. La notion de configuration composite semble donc "accessible" aux utilisateurs de la LPL malgré leur méconnaissance de nos technologies.

Par ailleurs, notre formalisme aide également au guidage par l'automatisation des actions. Comme nous l'avons présenté au dessus, plus de 94% des actions réalisées en moyenne pour créer un SDI sont automatisées, ce qui guide naturellement l'utilisateur vers une solution.

Enfin, les contextes sont pleinement exploités durant le processus de configuration : on constate qu'en moyenne pour un SDI créé, 12 contextes différents sont utilisés. Ce qui montre bien l'importance pour l'utilisateur d'avoir des perspectives différentes, en fonction des éléments qu'il configure. Le point faible reste cependant l'interface elle-même qui mériterait d'être améliorée afin de permettre une utilisation intensive par des utilisateurs non experts.

9.4.3.2 Garantir des temps de réponse et une empreinte mémoire raisonnable

La dynamicité du processus de configuration repose en grande partie sur les temps de réponse des outils : si les mécanismes de raisonnement paraissent trop long à l'utilisateur, celui-ci peut être amené à changer de tâche ou à perdre le fil de ce qu'il faisait. Nous avons mesuré le temps de résolution de notre algorithme de propagation durant les différents appels réalisés au cours des processus de configuration. Le Tableau 9.3 restitue les résultats ainsi obtenus.

On peut constater qu'en moyenne, le temps de propagation a été de 327,11 ms, ce qui semble être un temps de réponse très acceptable pour un humain [Dabrowski 11]. Cependant, si les chiffres sont raisonnables en moyenne, on peut constater de grandes disparités dans les durées mesurées, en fonction de la taille des chemins de propagation comme le montre d'ailleurs la valeur importante de l'écart type dans le Tableau 9.4. Ainsi, au dessus de 10 étapes de propagation, les temps deviennent relativement longs, supérieurs à 750 ms et jusqu'à 3,6 secondes pour le plus long chemin. Ces temps sont à relativiser, ce type de propagation n'ayant que très rarement lieu : il convient d'adapter les interfaces pour qu'elles invitent l'utilisateur à patienter lors d'un temps aussi long.

On peut s'interroger également sur l'apparente absence de corrélation entre les temps de propagation et la taille des chemins, pour des propagations de moins de 10 étapes. On a par exemple deux propagations de 10 étapes de 57 actions en moyenne qui s'exécutent en 38 ms en moyenne, contre 1 action de 9 étapes totalisant 78 actions qui s'exécute en plus de 2 secondes. Nous pensons que ces différences de résultats proviennent de plusieurs facteurs externes tels que les mécanismes d'allocation de la mémoire au sein des solveurs SAT ou encore l'état de la mémoire de la machine au moment de l'exécution des différents algorithmes de propagation.

Dans le cadre de notre approche, l'utilisateur a la possibilité de créer autant de contextes qu'il le souhaite. Or la création d'un nouveau contexte implique de créer autant de CPS que de concepts non bornés du modèle du domaine et pour chaque CPS de créer un état de configuration, une liste d'actions, etc. Ainsi, il nous semble important de déterminer si l'empreinte mémoire utilisée par l'implémentation de notre approche est raisonnable pour une utilisation normale.

Nous avons ainsi effectué des mesures sur la mémoire de notre machine en fonction de l'augmentation du nombre de contextes créés de 1 à 1000 par paliers. Nous avons pour cela testé la mémoire résidente de l'ordinateur avant l'initialisation de notre outil, puis nous avons artificiellement créé de nouveaux contextes.

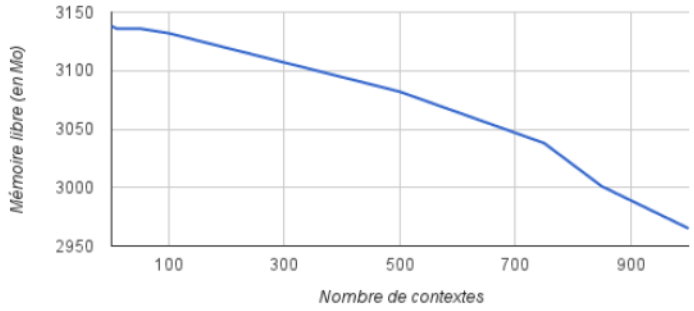


FIGURE 9.7 – Evolution de la mémoire disponible en fonction du nombre de contextes

Les résultats obtenus sont présentés dans la Figure 9.7. Nous pouvons constater que la mémoire disponible décroît linéairement en fonction du nombre de contextes créés : nous perdons 200 Mo de mémoire pour 1000 contextes créés, ce qui nous semble raisonnable dans le cadre d'une utilisation normale de notre système, par comparaison au nombre moyen de 12,5 contextes utilisés dans un processus de configuration standard (voir Tableau 9.2).

9.5 Conclusion

Nous avons présenté dans ce chapitre le processus de dérivation complet de la LPL YOURCAST en décrivant notamment le processus de génération de code aboutissant à un produit final. Nous avons ensuite discuté les différents résultats obtenus à partir d'un échantillon représentatif des processus de configuration réalisés dans cette LPL, en relation avec les objectifs que nous avons définis pour nos travaux de thèse.

Nous avons ainsi montré que la flexibilité du processus était assurée à la fois par notre modèle d'action et par l'algorithme de propagation qui permettait d'assurer la cohérence du processus, tout en permettant une absence d'ordre du workflow. Par ailleurs, l'utilisabilité de notre approche a été démontrée à partir des capacités de guidages permises par l'exploitation des

contextes et par les temps de réponse des opérations de propagations qui restent raisonnables malgré leur complexité.

CHAPITRE 10

CONCLUSION ET PERSPECTIVES

Sommaire

10.1 Conclusion

Nous avons posé au début de cette thèse deux défis concernant la définition et l'utilisation des LPL destinées à la production de systèmes-de-systèmes :

1. *Comment modéliser de manière cohérente la variabilité d'un système-de-systèmes au sein d'une LPL ?*
2. *Comment permettre un processus de configuration qui soit à la fois cohérent, flexible et utilisable dans le cadre d'une LPL complexe ?*

Comme nous l'avons montré, si les travaux de la littérature concernant les LPL sont nombreux, l'utilisation de LPL pour des systèmes-de-systèmes est un domaine de recherche relativement récent. Ainsi, peu de travaux portent sur les aspects de l'ingénierie du domaine liés à la représentation d'une variabilité complexe tout en offrant à un utilisateur non spécialiste la possibilité de configurer un produit en suivant une démarche flexible et cohérente.

Nous avons répondu à ces défis grâce à trois éléments de contribution que nous rappelons dans la suite. Les différents éléments de la contribution présentés dans cette thèse ont été précédemment publiés dans [Urli 14a, Urli 13, Urli 12a, Urli 12b].

10.1.1 Modélisation de la variabilité des LPL complexes

Nous avons défini un formalisme permettant notamment de modéliser sous la forme d'un modèle objets les différents concepts d'une LPL complexe et leurs relations. Dans notre formalisme les concepts sont liés à des FM afin de pouvoir représenter l'ensemble de leur variabilité sous forme de features. Par ailleurs notre modèle objet supporte la définition de multiplicités ainsi que l'expression de contraintes entre les FM afin d'exprimer différentes relations de dépendances entre les features.

Nous avons par ailleurs défini un algorithme permettant d'assurer la réalisabilité de l'ensemble de la LPL ainsi définie.

L'ensemble de ces travaux a été présenté dans le chapitre 5.

10.1.2 Réalisation de configurations dans le cadre d'un processus flexible et cohérent

Le formalisme que nous avons réalisé définit les différents concepts permettant de réaliser des actions de configuration au sein de la LPL. Il spécifie également des concepts permettant d'offrir des perspectives de configurations à l'utilisateur sous la forme de contextes et d'enregistrer l'ensemble des actions réalisées.

Par ailleurs, nous avons défini des algorithmes pour assurer la cohérence du processus de propagation quel que soit l'ordre des actions réalisées et d'autres pour annuler ces actions tout en continuant à assurer la cohérence.

Ces travaux ont été présentés dans le chapitre 6.

10.1.3 Définition et utilisation d'une LPL complexe

Enfin, nous avons réalisé une implémentation de notre formalisme et de nos algorithmes au travers d'une solution logicielle modulaire. Nous avons enrichi cette implémentation en réalisant des outils facilitant la modélisation des dépendances entre les FM (DSL de spécification

des règles et calcul automatique des règles inverses) et en réalisant une interface graphique de configuration pour les LPL complexes.

Nous avons présenté cette implémentation dans le chapitre 7.

10.2 Perspectives

Si cette thèse est l'aboutissement de trois années de recherches sur la façon de rendre les lignes de produits logicielles plus flexibles en terme de définition de la variabilité et en terme de configurations, elle a aussi permis d'ouvrir de nombreuses perspectives, tout d'abord de recherche mais également industrielles.

10.2.1 Modèle du domaine et contraintes

Si notre approche répond correctement aux objectifs que nous nous sommes fixés, le formalisme que nous avons défini pour le modèle du domaine reste très limité. Nous avons volontairement imposé des limites fortes sur le modèle du domaine afin de pouvoir garantir la cohérence de l'ensemble du processus.

Cependant, il serait judicieux d'étendre notre travail en considérant des hypothèses plus souples concernant le modèle du domaine, en permettant par exemple des multiplicités bornées supérieures à 1, ou en autorisant des associations réflexives.

Une autre limitation de notre modélisation de la variabilité concerne les règles de restrictions définies entre nos concepts : celles-ci ne s'expriment qu'entre des features. Il serait intéressant de pouvoir également exprimer des contraintes sur ou à partir d'éléments du modèle. Pouvoir, par exemple, exprimer que si 2 *pièces* ont été créées dans l'appartement alors une *ouverture* de type *fenêtre* doit nécessairement être créée. Cependant, étendre notre formalisme pour autoriser ce type de contraintes n'est pas trivial puisque toutes les propriétés de cohérence doivent continuer à être garanties.

10.2.2 Framework d'évolution

L'évolution des logiciels en général et plus spécifiquement des LPL est une problématique très importante du domaine. Nous avons eu l'occasion de travailler sur cette thématique dans le cadre du projet YOURCAST et y avons contribué au travers de trois publications : [Urli 14b, Urli 12a, Romero 13].

Ces travaux préliminaires nous laissent penser que l'intégration d'un framework d'évolution des LPL pourrait être envisageable avec notre approche. En effet, la séparation des préoccupations que nous offrons au niveau de la modélisation de la variabilité peut aider à gérer les problématiques d'évolution en considérant l'évolution d'un écosystème logiciel [Urli 14b].

Par ailleurs, notre algorithme de vérification de la réalisabilité permet de ne tester l'intégration que des nouveaux produits au sein de la LPL sans avoir à calculer la cohérence de la ligne toute entière. Enfin, notre modèle d'action nous offre tous les mécanismes nécessaires au rechargement des configurations précédemment enregistrées, à partir de la liste des actions utilisateur effectuées : les évolutions de la LPL pourraient alors être directement prises en compte en rejouant la configuration.

10.2.3 Collaboration, interface de configuration et gestion de préférences

Nous avons vu dans l'état de l'art que plusieurs travaux étudient les systèmes de configuration collaboratifs des LPL [Mendonça 08, Rabiser 10b].

Nous pensons que des mécanismes de synchronisation pourraient être mis en place dans l'implémentation de notre approche et une interface de configuration spécialement créée afin de supporter un tel processus. En effet, notre approche permet un processus de configuration sans ordre, tout en assurant sa cohérence. Nous pouvons donc assurer que, tant que deux actions n'ont pas lieu *en même temps*, alors le processus sera cohérent. Par ailleurs, nous pourrions autoriser la réalisation d'actions utilisateur en parallèle, en utilisant les contextes et notre algorithme d'annulation afin de gérer les conflits.

Nous souhaitons par ailleurs améliorer nos interfaces de configuration afin d'offrir une meilleure expérience utilisateur. L'étude ergonomique que nous avons réalisée sur nos interfaces graphiques a montré que celles-ci étaient mal adaptées à des utilisateurs non-experts, car elles manquaient d'explications et d'opérations de haut niveau pour l'utilisateur. Par exemple, si l'utilisateur peut créer une sous-configuration, il lui semble naturel de pouvoir la supprimer, même si, au sein de notre approche cela signifie réaliser une opération d'annulation de la création. Il est donc nécessaire de travailler cette interface pour l'adapter à nos utilisateurs.

Enfin, comme nous l'avons montré, la variabilité des systèmes-de-systèmes est très importante et les utilisateurs peuvent être rapidement perdus devant l'étendue de la variabilité, même en présence d'explications et de guidage pour les aider dans leur choix. Il nous semble ainsi pertinent de développer des mécanismes de gestion des préférences, exploitant des mécanismes d'apprentissage afin d'utiliser les configurations précédemment réalisées pour proposer à l'utilisateur des choix.

10.2.4 Framework de réalisation

Nous n'avons abordé dans nos travaux le processus de réalisation que pour le cas très concret de la LPL YOURCAST. Cependant, il est intéressant de constater que le processus utilise énormément d'informations apportées par la modélisation de la variabilité, telles que le modèle du domaine, les relations entre les concepts, les différents FM etc.

Ainsi, il nous semble pertinent d'étudier la possibilité de définir un framework générique de réalisation exploitant ces différentes informations et une structure spécifique d'assets.

10.2.5 SDI et perspective industrielle

Enfin, la réalisation de la LPL YOURCAST a mis en évidence un besoin au sein de l'industrie du web de l'utilisation d'outils tels que nous les avons définis. En effet, les LPL ont longtemps été cantonnées aux domaines des systèmes embarqués et intensifs alors que leur usage dans le domaine des systèmes d'informations aurait complètement sa place.

Nous prévoyons ainsi de créer une société afin d'exploiter les outils que nous avons développés dans le cadre des systèmes de diffusions d'informations, et pourquoi pas étendre ces outils à d'autres types de systèmes.

TABLE DES HYPOTHÈSES DE TRAVAIL

TABLE DES FIGURES

TABLE DES DÉFINITIONS ET PROPRIÉTÉS

LISTINGS

TABLE DES EXEMPLES

BIBLIOGRAPHIE

[Abbott 94] Kenneth R. Abbott & Sunil K. Sarin. *Experiences with Workflow Management : Issues for the Next Generation*. In Proceedings of the 1994 ACM conference on Computer Supported Cooperative Work (CSCW'94), pages 113–120, 1994.

[Acher 10a] Mathieu Acher, Philippe Collet, Philippe Lahire & Robert France. *Managing Multiple Software Product Lines Using Merging Techniques*. Technical report, ISRN I3S/RR, 2010.

[Acher 10b] Mathieu Acher, Philippe Collet, Philippe Lahire & Robert France. *Managing Variability in Workflow with Feature Model Composition Operators*. In Proceedings of the 9th International Conference on Software Composition (SC'10), volume LNCS of *Software Composition*, page 16, Malaga, Spain, 2010. Springer.

[Acher 11] M Acher, P Collet, P Lahire & R France. *A Domain-Specific Language for Managing Feature Models*. In Proceedings of the 2011 ACM Symposium on Applied Computing (SAC'11), pages 1333–1340. ACM, 2011.

[Acher 12] Mathieu Acher, Philippe Collet, Alban Gaignard, Philippe Lahire, Johan Montagnat & Robert France. *Composing Multiple Variability Artifacts to Assemble Coherent Workflows*. Software Quality Journal Special issue on Quality Engineering for Software Product Lines, vol. 20, no. 3-4, pages 689–734, 2012.

[Al-msie'deen 14] R. Al-msie'deen, M. Huchard, A.-D. Seriai, C. Urtado, S. Vauttier & A. Al-Khlifat. *Concept lattices : A representation space to structure software variability*. Proceedings of the 5th International Conference on Information and Communication Systems (ICICS'14), pages 1–6, April 2014.

[Arango 89] G. Arango. *Domain analysis : from art form to engineering discipline*. In Proceedings of the 5th international workshop on Software specification and design - IWSSD '89, volume 14, pages 152–159, New York, New York, USA, April 1989. ACM Press.

[Arboleda 13] Hugo Arboleda & Jean-Claude Royer. Model-Driven and Software Product Line Engineering. John Wiley & Sons, 2013.

[Asikainen 07] Timo Asikainen, Tomi Männistö & Timo Soininen. *Kumbang : A domain ontology for modelling variability in software product families*. Advanced Engineering Informatics, vol. 21, no. 1, pages 23–40, January 2007.

[Bagheri 10] Ebrahim Bagheri, Tommaso Di Noia, Azzurra Ragone & Dragan Gasevic. *Configuring Software Product Line Feature Models Based on Stakeholders' Soft and Hard Requirements*. Software Product Lines : Going Beyond, vol. 6287, pages 16–31, 2010.

[Bagheri 11] Ebrahim Bagheri & Dragan Gasevic. *Assessing the maintainability of software product line feature models using structural metrics*. Software Quality Journal, vol. 39, pages 1–39, 2011.

[Batory 05] Don Batory. *Feature models, grammars, and propositional formulas*. In Proceedings of the International Software Product Line Conference (SPLC'05), pages 7–20, 2005.

[Benavides 10] David Benavides, Sergio Segura & Antonio Ruiz-Cortés. *Automated analysis of feature models 20 years later : A literature review*. Information Systems, vol. 35, no. 6, pages 615–636, September 2010.

[Berger 13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki & Andrzej Wąsowski. *A survey of variability modeling in industrial practice*. Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '13, page 1, 2013.

[Blay-Fornarino 12] Mireille Blay-Fornarino, Philippe Collet, Laurence Duchien, Philippe Renevier, Daniel Romero, Philippe Salvan & Simon Urli. *Cahier des charges fonctionnel : Diffusion d'informations dans le cadre de grands rassemblements gérés par l'ERP intr@ssoc*. Livrable I-2.2.1, I3S, Sophia Antipolis, France, 2012.

[Boardman 06] John Boardman & B Sauser. *System of Systems-the meaning of of*. In System of Systems Engineering, pages 118–123, 2006.

[Bosch 02] Jan Bosch. *Maturity and evolution in software product lines : Approaches, artefacts and organization*. In Proceedings of the International Software Product Line Conference (SPLC'02), pages 257–271, 2002.

[Bosch 10] Jan Bosch. *Toward compositional software product lines*. Software, IEEE, vol. 27, no. 3, pages 29–34, 2010.

[Botterweck 13] Goetz Botterweck. *Variability and Evolution in Systems of Systems*. Electronic Proceedings in Theoretical Computer Science, vol. 133, no. AiSoS, pages 8–23, November 2013.

[Bąk 11] K Bąk, K. Czarnecki & A Wąsowski. *Feature and meta-models in Clafer : mixed, specialized, and coupled*. In Proceedings of the 3rd International Conference on Software Language Engineering (SLE'11), pages 102–122. Springer, 2011.

[Campbell 90] Grady H. Campbell, Stuart R. Faulk & David M. Weiss. *Introduction to Synthesis*. Technical report, Software productivity Consortium, 1990.

[Classen 08] Andreas Classen, Patrick Heymans & Pierre-Yves Schobbens. What's in a Feature : A Requirements Engineering Perspective, volume 4961, pages 16–30. Fundamental Approaches to Software Engineering (FASE), 2008.

[Classen 09] Andreas Classen, Arnaud Hubaux & Patrick Heymans. *A Formal Semantics for Multi-level Staged Configuration*. Technical report, 2009.

[Classen 11] Andreas Classen, Quentin Boucher & Patrick Heymans. *A text-based approach to feature modelling : Syntax and semantics of TVL*. Science of Computer Programming, vol. 76, no. 12, pages 1130–1143, December 2011.

[Clements 02] Paul Clements & Linda Northrop. *Software Product Lines : Practices and Patterns*. 2002.

[Czarnecki 00] Krzysztof Czarnecki & Ulrich Eisenecker. Generative Programming : Methods, Tools, and Applications. 2000.

[Czarnecki 02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger & Ulrich Eisenecker. *Generative Programming for Embedded Software : An Industrial Experience Report*. In Proceedings of the 1st conference on Generative Programming and Component Engineering (GPCE), volume 2487, pages 156–172, 2002.

[Czarnecki 05a] Krzysztof Czarnecki, Simon Helsen & Ulrich Eisenecker. *Formalizing cardinality-based feature models and their specialization*. Software Process Improvement and Practice, vol. 10, pages 7–29, 2005.

[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.

[Czarnecki 12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser & Klaus Schmid. *Cool Features and Tough Decisions : A Comparison of Variability Modeling Approaches*. In Proceedings of the International Workshop on Variability Modelling of Software intensive Systems (VaMoS), pages 173–182. ACM Press, ACM Press, 2012.

[Dabrowski 11] Jim Dabrowski & Ethan V. Munson. *40 Years of Searching for the Best Computer System Response Time*. Interacting with Computers, vol. 23, no. 5, pages 555–564, September 2011.

[Davis 87] Stanley M. Davis. Future Perfect. Addison-Wesley, Boston, Massachusetts, 1987.

[Deelstra 05] Sybren Deelstra, Marco Sinnema & Jan Bosch. *Product derivation in software product families : a case study*. Journal of Systems and Software, vol. 74, no. 2, pages 173–194, January 2005.

[Dhungana 10] Deepak Dhungana, Paul Grünbacher & Rick Rabiser. *The DOPLER meta-tool for decision-oriented variability modeling : a*

multiple case study. Automated Software Engineering, vol. 18, no. 1, pages 77–114, November 2010.

[Dhungana 11] Deepak Dhungana, Rick Rabiser, Paul Grunbacher, Dominik Seichter, Goetz Botterweck, David Benavides & José A. Galindo. *Configuration of multi product lines by bridging heterogeneous variability modeling approaches*. In Proceedings of the 15th International Software Product Line Conference (SPLC'11), pages 120–129, 2011.

[Dhungana 13] Deepak Dhungana, Rick Rabiser, Paul Grunbacher, Dominik Seichter, Goetz Botterweck, David Benavides & José A. Galindo. *Integrating heterogeneous variability modeling approaches with invar*. In Proceedings of the 7th International Workshop on Variability Modelling of Software intensive Systems (VaMoS'13), pages 32–37, 2013.

[Diskin 13] Zinovy Diskin, Aliakbar Safilian & Tom Maibaum. *Modeling product lines with kripke structures and modal logic*. Technical Report October, 2013.

[Djisktra 76] Edsger Wybe Djisktra. A discipline of programming. 1976.

[Elsner 11] Christoph Elsner, Daniel Lohmann & Wolfgang Schröder-Preikschat. *An infrastructure for composing build systems of software product lines*. In Proceedings of the 15th International Software Product Line Conference (SPLC'11), page 8, 2011.

[Faltings 98] B. Faltings & E.C. Freuder. *Configuration [Guest Editor's Introduction]*. IEEE Intelligent Systems and their Applications, vol. 13, no. 4, pages 32–33, July 1998.

[Ferreira Filho 12] Joao Bosco Ferreira Filho, Olivier Barais, Benoit Baudry & Jerome Le Noir. *Leveraging variability modeling for multi-dimensional Model-driven Software Product Lines*. In Proceedings of the 3rd International Workshop on Product LinE Approaches in Software Engineering (PLEASE), pages 5–8. Ieee, June 2012.

[Friess 07] Wolfgang Friess, Julio Sincero & Wolfgang Schroeder-preikschat. *Modelling Compositions of Modular Embedded Software Product Lines*. In Proceedings of the 25th conference on IASTED International Multi-Conference : Software Engineering, pages 224–228, 2007.

[Gaffney 92] J. E. Gaffney & R. D. Cruickshank. *A general economics model of software reuse*. In Proceedings of the 14th international conference on Software engineering - ICSE '92, pages 327–337, New York, New York, USA, June 1992. ACM Press.

[Hartmann 08] Herman Hartmann & Tim Trew. *Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains*. In Proceedings of the International Software Product Line Conference (SPLC'08), pages 12–21. IEEE, 2008.

[Helferich 07] Andreas Helferich, Georg Herzwurm, Stefan Jesse & Martin Mikusz. *Software product lines, service-oriented architecture and frameworks : worlds apart or ideal partners ?* Trends in Enterprise Application Architecture, pages 187–201, 2007.

[Holl 12] Gerald Holl, Paul Grünbacher & Rick Rabiser. *A systematic review and an expert survey on capabilities supporting multi product lines*. Information and Software Technology, vol. 54, no. 8, pages 828–852, August 2012.

[Hopcroft 73] John Hopcroft & Robert Tarjan. *Efficient algorithms for graph manipulation*. Communications of the ACM, vol. 16, no. 6, 1973.

[Hubaux 09] Arnaud Hubaux, Andreas Classen & Patrick Heymans. *Formal Modelling of Feature Configuration Workflows*. In Proceedings of the 13th International Software Product Line Conference (SPLC'09), pages 221–230, 2009.

[Hubaux 12a] Arnaud Hubaux, D Jannach, C Drescher, L Murta, T Mannisto, Patrick Heymans, Krzysztof Czarnecki, Thanh Hai Nguyen & M Zanker. *Unifying Software and Product Configuration : A Research Roadmap*. In Proceedings of the Workshop on Configuration (ConfWS), Montpellier, France, 2012.

[Hubaux 12b] Arnaud Hubaux, Yingfei Xiong & Krzysztof Czarnecki. *A user survey of configuration challenges in Linux and eCos*. Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '12, pages 149–155, 2012.

[Hubaux 13] Arnaud Hubaux, Patrick Heymans, Pierre Yves Schobbens, Dirk Deridder & Ebrahim Khalil Abbasi. *Supporting multiple perspectives in feature-based configuration*. Software and Systems Modeling, vol. 12, pages 641–663, 2013.

[Jansen 09] Slinger Jansen, Anthony Finkelstein & Sjaak Brinkkemper. *A sense of community : A research agenda for software ecosystems*. In Proceedings of the 31st International Conference on Software Engineering (ICSE'09), pages 187–190. Ieee, 2009.

[John 09] Isabel John & Michael Eisenbarth. *A decade of scoping : a survey*. In Proceedings of the 13th International Software Product Line Conference (SPLC'09), pages 31–40, 2009.

[Junker 06] Ulrich Junker. *Configuration*. In Francesca Rossi, Peter van Beek & Toby Walsh, editeurs, Handbook of Constraint Programming (Foundations of Artificial Intelligence), chapitre 24. Elsevier Science Inc., October 2006.

[Kang 90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak & A. Spencer Peterson. *Feature-Oriented Domain Analysis (FODA) Feasability Study*. Technical Report 1, 1990.

[Kelsen 10] Keith Kelsen. Unleashing the power of digital signage - Content strategies for the 5th screen. Elsevier Inc., focal pres edition, 2010.

[Kiczales 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier & John Irwin. Aspect-Oriented Programming. 1997.

[Lacomme 03] Philippe Lacomme, Christian Prins & Marc Sevaux. Algorithmes de Graphes. 2003.

[Leitner 11] Andrea Leitner, Christian Steger, Christian Kreiner, Roland Mader & Reinhold Weiß. *Towards Multi-modeling for Domain Description*. In Proceedings of the 15th International Software Product Line Conference (SPLC'11), page 6, 2011.

[Maier 98] MW Maier. *Architecting principles for systems-of-systems*. Systems Engineering, 1998.

[Mendonca 09] Marcilio Mendonca, A Wąsowski & K Czarnecki. *SAT-based analysis of feature models is easy*. In Proceedings of the 13th International Software Product Line Conference (SPLC'09), pages 231–240, 2009.

[Mendonça 08] Marcílio Mendonça, Donald Cowan, William Malyk & Toacy Oliveira. *Collaborative product configuration : Formalization and efficient algorithms for dependency analysis*. Journal of Software, vol. 3, no. 2, pages 69–82, 2008.

[Metzger 14] Andreas Metzger & Klaus Pohl. *Software product line engineering and variability management : achievements and challenges*. Proceedings of the on Future of Software Engineering - FOSE 2014, pages 70–84, 2014.

[Michel 11] Raphael Michel, Andreas Classen, Quentin Boucher & Arnaud Hubaux. *A Formal Semantics for Feature Cardinalities in Feature Diagrams*. In Proceedings of the 5th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'11), pages 82–89. ACM, 2011.

[Morin 09] Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais & Gilles Vanwormhoudt. *Weaving Variability into Domain Metamodels*. Model driven engineering languages and systems, vol. 5795, pages 690–705, 2009.

[Nierstrasz 92] Oscar Nierstrasz, Simon Gibbs & Dennis Tsichritzis. *Component-oriented software development*. Communications of the ACM, vol. 35, no. 9, pages 160–165, September 1992.

[Northrop 06] Linda Northrop, Peter Feiler, Richard Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Douglas Schmidt, Kevin Sullivan & Kurt Wallnau. Ultra-Large-Scale Systems - The Software Challenge of the Future. 2006.

[OMG Revised Submission 12] OMG Revised Submission. *Common Variability Language (CVL)*. Technical Report Cvl, OMG, 2012.

[Parnas 76] David L. Parnas. *On the Design and Development of Program Families*. IEEE Transactions on Software Engineering, vol. SE-2, no. 1, 1976.

[Pohl 05] Klaus Pohl, Günter Böckle & Frank van der Linden. Software Product Line Engineering : Foundations, Principles and Techniques. Springer, 2005.

[Possompès 11] Thibaut Possompès, François Briant, Christophe Dony, Marianne Huchard & Chouki Tibermacine. *Diagramme de Features, adaptation par transformation dans le contexte des bâtiments intelligents*. In Journée Ligne de Produits (JLdP'11), Paris, France, 2011.

[Possompès 13] Thibaut Possompès. *Configuration par modèle de caractéristiques adapté au contexte pour les lignes de produits logiciels*. PhD thesis, Université Montpellier 2, 2013.

[Rabiser 09] Rick Rabiser, Reinhard Wolfinger & Paul Grünbacher. *Three-level Customization of Software Products Using a Product Line Approach*. In Proceedings of the Hawaii International Conference on System Sciences, pages 1–10, 2009.

[Rabiser 10a] Rick Rabiser, Paul Grünbacher & Deepak Dhungana. *Requirements for product derivation support : Results from a systematic literature review and an expert survey*. Information and Software Technology, vol. 52, no. 3, pages 324–346, March 2010.

[Rabiser 10b] Rick Rabiser, Paul Grünbacher & Gerald Holl. *Improving Awareness during Product Derivation in Multi-User Multi Product Line Environments*. In Proceedings of the International Workshop on Automated Tailoring and Configuration of Applications (ACoTA), pages 1–5, 2010.

[Rashid 11] Awais Rashid, Jean-Claude Royer & Andreas Rummler. Aspect-Oriented, Model-Driven Software Product Lines : The AMPLE Way. Cambridge edition, 2011.

[Reiser 06] Mark-oliver Reiser & Matthias Weber. *Managing Highly Complex Product Families with Multi-Level Feature Trees*. In Proceedings of the International Requirements Engineering Conference (RE), pages 149–158, 2006.

[Riebisch 02] M Riebisch, K Böllert, D Streitferdt & I Philippow. *Extending Feature Diagrams with UML Multiplicities*. In Proceedings of the 6th World Conference on Integrated Design & Process Technology, pages 1–7, 2002.

[Romero 13] Daniel Romero, Simon Urli, Clément Quinton, Mireille Blay-Fornarino, Philippe Collet, Laurence Duchien & Sébastien Mosser. *SPLEMMA : a generic framework for controlled-evolution of software product lines*. In Proceedings of the International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE), pages 59–66, 2013.

[Rosenmüller 08] Marko Rosenmüller, Norbert Siegmund, Christian Kästner & Syed Saif Ur Rahman. *Modeling dependent software product lines*. In Proceedings of the Workshop on Modularization,

Composition and Generative Techniques for Product Line Engineering (McGPLE), pages 13–18, 2008.

[Rosenmüller 10] Marko Rosenmüller & Norbert Siegmund. *Automating the configuration of multi software product lines*. In Variability Modelling of Software intensive Systems VaMoS, 2010.

[Schirmeier 09] Horst Schirmeier & Olaf Spinczyk. *Challenges in software product line composition*. In Proceedings of the Hawaii International Conference on System Sciences, pages 1–7, 2009.

[Schobbens 07] Pierre Yves Schobbens, Patrick Heymans, Jean Christophe Trigaux & Yves Bontemps. *Generic semantics of feature diagrams*. Computer Networks, vol. 51, pages 456–479, 2007.

[Schroeter 12] Julia Schroeter, Malte Lochau & Tim Winkelmann. *Multi-perspectives on Feature Models*. In Proceedings of the Model Driven Engineering Languages and Systems Conference (MODELS'12), pages 252–268, 2012.

[Urli 12a] Simon Urli, Mireille Blay-fornarino & Philippe Collet. *Using Composite Feature Models to Support Agile Software Product Line Evolution*. In Proceedings of the 6th International Workshop on Models and Evolution (ME'12), pages 21–26, 2012.

[Urli 12b] Simon Urli, Guillaume Perez, Heytem Zitoun, Mireille Blay, Philippe Collet & Philippe Renevier-gonin. *Vers des interfaces graphiques flexibles de configurations*. In Proceedings of the Journées sur les Lignes De Produits Logiciels (JLDP'12), page 5, 2012.

[Urli 13] Simon Urli, Sébastien Mosser, Mireille Blay-Fornarino & Philippe Collet. *How to Exploit Domain Knowledge in Multiple Software Product Lines ?* In Proceedings of the 4th International Workshop on Product LinE Approaches in Software Engineering (PLEASE'13), PLEASE, page 4, San Francisco, USA, 2013. ACM.

[Urli 14a] Simon Urli, Mireille Blay-fornarino & Philippe Collet. *Handling Complex Configurations in Software Product Lines : a Tooled Approach*. In Proceedings of the 18th International Software Product Line Conference (SPLC'14), pages 112–121, Florence, Italy, 2014.

[Urli 14b] Simon Urli, Mireille Blay-fornarino, Philippe Collet, Sébastien Mosser & Michel Riveill. *Managing a Software Ecosystem Using a Multiple Software Product Line : a Case Study on Digital Signage Systems*. In Proceedings of the Euromicro Conference series on Software Engineering and Advanced Applications (SEAA'14), pages 344–351, 2014.

[van Gurp 01] J. van Gurp, J. Bosch & M. Svahnberg. *On the notion of variability in software product lines*. Proceedings Working IEEE/IFIP Conference on Software Architecture, 2001.

[van Ommering 00] Rob van Ommering. *Beyond Product Families : Building a Product Population ?* In Proceedings of the International Workshop on Software Architectures for Product Families, pages 187–198, 2000.

[van Ommering 02] Rob van Ommering & Jan Bosch. *Widening the Scope of Software Product Lines - From Variation to Composition*. In Software Product Lines, pages 31–52. Springer, 2002.

[Wilf 94] Herbert S Wilf. Algorithms and Complexity. 1994.

RÉSUMÉ

Résumé La nécessité de produire des logiciels de qualité en adéquation avec les besoins spécifiques du marché a conduit à l'émergence de nouvelles approches de développements telles que les *Lignes de Produits Logiciels* (LPL). Cependant pour répondre aux exigences croissantes des nouveaux systèmes informatiques, il convient aujourd'hui d'envisager la production de ces systèmes comme des compositions d'un grand nombre de systèmes interconnectés que l'on nomme aujourd'hui des *systèmes-de-systèmes*. En terme de lignes de produits, il s'agit de supporter la modularité et la très grande variabilité de ces systèmes, aussi bien du point de vue de la définition des sous-systèmes, que du point de vue de leur composition tout en garantissant la viabilité des systèmes construits.

Pour supporter la construction et l'utilisation de lignes de produits logiciels complexes, nous proposons une nouvelle approche basée sur *(i)* la définition du modèle du domaine de la ligne, *(ii)* la formalisation de la variabilité des éléments du domaine par des feature models (FM) et *(iii)* l'expression des dépendances entre ces différents FM. Pour maîtriser la complexité de telles lignes nous avons complété cette approche de modélisation par d'une part, des algorithmes visant à assurer la cohérence des lignes ainsi modélisées et d'autre part, la conception d'un processus de configuration des produits logiciels complexes garantissant la cohérence des produits sans imposer d'ordre dans les choix utilisateurs et en autorisant l'annulation des choix.

Cette thèse présente une formalisation de ces travaux démontrant ainsi les propriétés attendues de ces LPL comme la maîtrise de la complexité de la ligne par des algorithmes incrémentiels exploitant la topologie du modèle du domaine, la définition formelle et la preuve de la flexibilité du processus de configuration ou les notions de cohérence du processus lui-même. Sur cette base bien fondée, nous proposons une implémentation possible intégrant des éléments additionnels pour supporter le développement de telles lignes tels qu'une interface graphique de configuration générique qui nous a servi de support aux expérimentations. Nous validons nos travaux sur une LPL dédiée à un système-de-systèmes de portée industrielle pour la production de systèmes de diffusion d'informations.

Abstract The necessity of producing high quality softwares and the specific software market needs raise new approaches such as *Software Product Lines* (SPL). However in order to satisfy the growing requirements of new information systems, we need to consider those systems as a composition of many interconnected sub-systems called *systems-of-systems*. As a SPL, it implies to support the modularity and the large variability of such systems, from the definition of sub-systems to their composition, ensuring the consistency of final systems.

To support design and usage of such a complex SPL, we propose a new approach based on *(i)* the definition of a SPL domain model, *(ii)* the formalization of variability using feature models (FM) and *(iii)* the representation of dependencies between those different FM. In order to manage the complexity of this SPL we complete our approach by in one hand algorithms ensuring the consistency of the SPL and on the other hand the definition of a configuration process which guarantees the consistency of products without imposing order in user choices and authorizing to cancel any choice.

This thesis presents a formalization of these works and demonstrates the expected properties of those SPL, like the control of the product line consistency with incremental algorithms exploiting the domain model topology, the formal definition and the proof of the configuration process flexibility, and the consistency concepts of the process itself. On these basis, we propose a first implementation containing additionnal elements in order to support the design and the use of the SPL like a generic graphical user interface dedicated to the configuration process, which helps us during our experiments. We validate our works on a SPL dedicated to an industrial scale system-of-systems for producing digital signage systems.