L'analyse formelle de concepts : un cadre structurel pour l'étude de la variabilité de familles de logiciels

Jessie Carbonnel

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITE DE MONTPELLIER

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM UMR 5506

L'analyse formelle de concepts : un cadre structurel pour l'étude de la variabilité de familles de logiciels

Présentée par Jessie CARBONNEL

Le 29 octobre 2018

Sous la direction de Marianne HUCHARD et Clémentine NEBUT

Devant le jury composé de

Jean-Marc Jézéquel, Professeur à l'Université de Rennes 1, IRISA, Rennes	Rapporteur
Amedeo Napoli, Directeur de Recherche CNRS, LORIA, Nancy	Rapporteur
Mathieu Acher, Maître de Conférences à l'Université de Rennes 1, IRISA, Rennes	Examinateur
Karell Bertet, Maître de Conférences HDR à l'Université de La Rochelle, L3I, La Rochelle	Examinatrice
Anne Laurent, Professeur à l'Université de Montpellier, LIRMM, Montpellier	Présidente du jury
André Miralles, Chercheur HDR IRSTEA (retraité), TETIS, Montpellier	Invité
Clémentine Nebut, Maître de Conférences à l'Université de Montpellier, LIRMM, Montpellier	Co-encadrante
Marianne Huchard, Professeur à l'Université de Montpellier, LIRMM, Montpellier	Directrice de thèse

UNIVERSITÉ DE MONTPELLIER

Table des matières

Résumé

Des **familles de logiciels similaires** émergent fréquemment dans les entreprises, lorsque de nouveaux logiciels sont créés en réutilisant des clones de logiciels existants, qui sont ensuite enrichis ou dépouillés de fonctionnalités pour suivre de nouvelles exigences. Avec le temps, ces variantes se multiplient et se complexifient, et il devient difficile de les maintenir, de les faire évoluer, ou bien de s'en servir pour en créer de nouvelles.

L'**ingénierie des lignes de produits logiciels** regroupe un ensemble de méthodes visant à faciliter le développement et la gestion de telles collections de logiciels similaires. C'est un paradigme de développement basé sur la mise en place d'une architecture logicielle générique structurant un ensemble d'artefacts réutilisables, depuis laquelle différentes variantes logicielles peuvent être dérivées automatiquement. Documenter la **variabilité** est le point central de ce paradigme, qui consiste à définir ce qui est commun, ce qui varie, et comment ces variations interviennent au sein de la famille de logiciels. On la représente à travers des **modèles de variabilité**, qui servent de supports à la grande majorité des processus propres à l'ingénierie des lignes de produits.

La **migration complète ou partielle** de ces familles de logiciels vers des approches de type lignes de produits permet la **simplification de leur exploitation**. La rétro-ingénierie, la modélisation et la gestion de la variabilité sont reconnues comme une phase cruciale et ardue de cette migration. Par conséquent, de nombreuses approches ont été proposées pour étudier des descriptions de familles de logiciels dans ce but. Ces approches sont majoritairement construites pour répondre à des problématiques spécifiques (e.g., rétro-ingénierie de modèles, détection de caractéristiques obsolètes, dérivation de configurateurs), en se fondant sur une étude ad-hoc de la variabilité à travers des structures et des techniques disparates. Plusieurs d'entre elles s'appuient sur l'**analyse formelle de concepts**, un cadre mathématique de groupement hiérarchique qui organise un ensemble d'objets et leurs descriptions dans une structure canonique mettant naturellement en évidence leurs aspects communs et variables. Principalement utilisé pour l'extraction d'informations sous forme de relations logiques et l'identification de caractéristiques dans du code, ce cadre n'a cependant pas été plus longuement exploré. Dans ce manuscrit, nous approfondissons l'étude de ce formalisme et montrons ses vertus en termes de représentation et de support aux opérations de gestion de la variabilité. De plus, l'analyse des approches existantes pour accompagner la migration montre que beaucoup d'entre elles peuvent être ramenées à l'analyse formelle de concepts, qui offre un cadre méthodique ayant de solides bases théoriques pour soutenir ces opérations. **La thèse que je défends dans ce manuscrit est que l'analyse formelle de concepts, plus qu'un outil, offre un véritable cadre structurel et réutilisable à l'étude de la variabilité des familles de produits.**

Dans un premier temps, nous établissons un panorama des informations sur la variabilité qui sont mises en évidence grâce à ce formalisme, et discutons de son spectre d'applicabilité. Nous étudions les points communs entre les structures conceptuelles produites par l'analyse formelle de concepts et les modèles de variabilité. Dans un second temps, nous élargissons notre champ d'étude aux informations plus complexes définies par des modèles de variabilité qui ont été étendus pour en améliorer l'expressivité, et dont la rétro-ingénierie est encore peu étudiée à ce jour. Nous montrons comment certaines propriétés de l'analyse formelle de concepts permettent de généraliser son utilisation à des descriptions de variantes plus complexes, et étudions son application pour la manipulation d'**attributs multivalués** et de **cardinalités**, en complément des caractéristiques booléennes traditionnelles. Enfin, nous illustrons l'utilisation originale de ces structures conceptuelles comme support à des opérations de conception et de recherche d'informations, qui permettent de limiter l'intervention d'un expert dans le processus de modélisation.

Chapitre 1

Introduction

Préambule

Dans cette introduction, nous présentons d'abord le contexte de nos travaux, qui s'inscrivent dans le domaine de l'ingénierie des lignes de produits logiciels. Plus précisément, nous proposons d'étudier l'analyse formelle de concepts, un cadre mathématique structurel pour l'analyse de données, afin d'encadrer la rétro-ingénierie des lignes de produits logiciels. Nous abordons ensuite les motivations et les objectifs de nos travaux. Enfin, nous présentons le plan de ce manuscrit et nous détaillons les inspirations et les collaborations qui sont à l'origine de ces travaux ainsi que les contributions réalisées.

Sommaire

1.1 Contexte

Une famille de produits logiciels (FPL) [Par76] désigne une collection de systèmes logiciels destinés à des clients et présentant des similarités non négligeables. De telles familles proviennent de pratiques telles que le *clone-and-own* (cloner et s'approprier) [DRB+13], qui consiste à copier le code source et autres artefacts d'un système logiciel existant et à le modifier pour en faire un nouveau. La copie du système existant permet d'éviter de redévelopper les artefacts qui sont communs avec la copie modifiée. Cela représente donc une pratique ad-hoc et souvent non organisée de la réutilisation logicielle. Sur le long terme, il est de plus en plus difficile de gérer de telles collections de variantes logicielles sans un cadre approprié [Kru01].

Il existe cependant des moyens de faciliter l'exploitation des FPLs malgré leur nombre et leur complexité grandissants.

1.1.1 L'ingénierie des lignes de produits logiciels

L'ingénierie des lignes de produits logiciels [PBvdL05] regroupe un ensemble de méthodologies pour créer, manipuler, maintenir et faire évoluer une FPL. C'est un paradigme de développement basé sur une architecture logicielle générique et un ensemble d'artefacts réutilisables. L'architecture regroupe les points communs à toutes les variantes logicielles et représente donc le cœur invariable de la FPL. Elle présente de plus des points de variation, permettant d'y ajouter de manière prescrite des artefacts réutilisables préalablement définis. Pour des exigences données, on peut combiner les artefacts et étendre les fonctionnalités de base de l'architecture pour former une variante logicielle à part entière. L'architecture générique, les artefacts réutilisables et l'ensemble de variantes logicielles qui peuvent en être dérivées forment une **ligne de produits logiciels** (LPL). Ce paradigme de développement est donc régi par deux principes : la réutilisation systématique, mise en œuvre à travers l'architecture logicielle générique permettant l'organisation de la réutilisation des différents artefacts et la personnalisation de masse, rendue possible grâce aux points de variation de l'architecture permettant d'y ajouter différentes combinaisons d'artefacts répondant à des exigences.

Le point considéré comme central de l'ingénierie des LPLs est la modélisation de la variabilité [CGR+12]. C'est une tâche qui consiste à représenter les éléments communs, les éléments variables et la façon dont ces éléments varient d'un produit logiciel à l'autre de la FPL. Traditionnellement, la variabilité s'exprime en termes de caractéristiques [KLD02, ABKS13], qui sont des éléments de haut niveau servant à distinguer les produits logiciels et devant être compréhensibles par tous les intervenants. Les modèles de caractéristiques (*feature models* en anglais) [KCH+90] sont les modèles de variabilité les plus populaires de l'approche orientée caractéristiques. Ces modèles sont utilisés comme support à la plupart des tâches de l'ingénierie des LPLs, comme la représentation d'informations [CKK06], mais aussi pour définir le périmètre de la LPL, son évolution et sa maintenance [LP07], la réalisation d'opérations de conception [ACLF10b, ACLF10a] ou bien la dérivation de produits [JSvGB04].

1.1.2 La rétro-ingénierie des lignes de produits logiciels

La stratégie d'adoption *proactive* de ce paradigme (i.e., construction ex nihilo de la LPL) se divise en deux grands processus successifs [PBvdL05, Kru02]. Le premier processus a pour objectif de développer et de documenter l'architecture générique et les artefacts réutilisables. Le second cherche à mettre en place des mécanismes pour identifier les exigences requises par un utilisateur et dériver le produit logiciel correspondant. Bien que cette

stratégie d'adoption semble être la manière naturelle de construire une LPL, elle n'est cependant pas la plus répandue dans les entreprises développant des LPLs [BRN+13]. C'est la stratégie d'adoption dite *extractive* qui semble être la plus populaire : celle-ci consiste à construire une LPL basée sur un ensemble de variantes existantes, par induction. En effet, développer des systèmes logiciels similaires hors d'un cadre tel que celui de l'ingénierie des LPLs permet d'obtenir des produits fonctionnels plus rapidement et aux différents intervenants de se familiariser avec la variabilité de la FPL. La difficulté de maintenir ces produits lorsque ceux-ci sont de plus en plus nombreux et complexes donne lieu à une transition vers des approches de type LPLs [Kru01, FV03, RPK10].

Naturellement, de nombreux travaux réalisés dans le contexte de la migration vers des LPLs se sont orientés vers la rétro-ingénierie des modèles de caractéristiques à partir de descriptions de variantes logicielles existantes, du fait de leur importance dans ce paradigme.

À notre connaissance, les méthodes proposées pour la synthèse de modèles de caractéristiques sont souvent définies de manière fonctionnelle. Ce sont des méthodes ad-hoc, reposant sur des représentations intermédiaires de l'information, construites pour établir un cadre provisoire dans lequel réaliser une opération définie. Ces méthodes existent indépendamment d'un cadre général de rétro-ingénierie, comme en témoigne la diversité des approches utilisées, incluant la logique propositionnelle [CW07, SRA+14, ACP+12], des représentations intermédiaires de la variabilité [CW07, SRA+14], des métaheuristiques [LLG+15, LLE14] ou encore des algorithmes dédiés [HLE11, HLE13]. Des outils tels que `But4Reuse` (pour *bottom-up technologies for reuse*) [MZB+17] regroupent différentes tâches nécessaires à la rétro-ingénierie des LPLs. C'est un cadre permettant de combiner différents processus pouvant être basés sur des formalismes et des approches très différents. Il a l'avantage de regrouper un large panel d'outils et d'approches existants, tout en facilitant leur jonction.

Parmi les méthodes visant à faciliter et à automatiser la migration, certaines ont en commun leur utilisation de l'analyse formelle de concepts [RPK11, XXJ12, AMSH+13, AMHS+14, SSS17]. L'analyse formelle de concepts (AFC) [GW99] est un cadre mathématique structurel pour l'analyse de données, la gestion d'informations et la représentation des connaissances. L'AFC réalise un groupement hiérarchique à partir de descriptions objets × attributs, représentant un ensemble fini d'objets formels décrits par des attributs formels. Ces groupes sont appelés concepts formels, un concept représentant un ensemble maximal d'objets partageant un ensemble maximal d'attributs ; tous les concepts formels extraits des descriptions initiales sont ensuite partiellement ordonnés par inclusion ensembliste pour former les structures de bases de l'AFC, qui s'apparentent à des hiérarchies de spécialisation canoniques. Les concepts formels décrivent des caractéristiques fondamentales de la recherche d'information : les structures conceptuelles de l'AFC ont de ce fait été redécouvertes plusieurs fois par des chercheurs différents [Pri06]. Elles organisent naturellement les objets en fonction des attributs qu'ils partagent et mettent ainsi en évidence ce qui est commun et ce qui varie en termes d'attributs : cela fait d'elles des structures naturelles de représentation de la variabilité [AMdSH+14].

1.2 Motivations et objectifs

Dans ce qui suit, nous expliquons pourquoi nous étudions l'AFC dans le contexte de la migration vers des LPLs et dans quels buts.

1.2.1 Motivations

On peut facilement établir un rapprochement entre l'approche de modélisation orientée caractéristiques, omniprésente dans l'ingénierie des LPLs et les descriptions objets × attributs de l'AFC. Cette proximité dans la représentation des descriptions des données initiales a certainement facilité son adoption dans le contexte de la migration [RPK11, XXJ12, SSS17] et de l'ingénierie des LPLs en général [LP07, NE09] : considérer les configurations valides comme des objets formels et leurs caractéristiques comme des attributs formels permet d'utiliser les structures de base de l'AFC comme des structures de représentation de la variabilité en termes de caractéristiques.

De plus, les structures de l'AFC sont des représentations qui incluent différentes représentations de la variabilité utilisées dans la littérature, telles que les graphes d'implications binaires, les graphes de mutex, les diagrammes de décision binaires ou encore les formules propositionnelles (où les relations objets × attributs sont des relations modèles × variables propositionnelles). Leur construction se base de plus sur de fortes bases théoriques, contrairement aux métaheuristiques et aux algorithmes dédiés.

Parce que structurel, ce cadre peut donc être facilement réutilisé et étendu, contrairement aux méthodes de représentation de la variabilité actuelles. Ses nombreux domaines d'application (par exemple, la linguistique [Pri05], l'apprentissage automatique [Kuz01], la recherche d'information [CN15] ou l'ingénierie logicielle [TCBE05]) et les diverses tâches qu'il permet de mettre en place témoignent de sa réutilisabilité. On trouve par ailleurs une collection d'extensions diverses, telles que l'analyse temporelle de concepts [Wol01], l'analyse relationnelle de concepts [RHHNV13], l'analyse logique de concepts [FR00], ou encore les structures de patterns [GK01] ; ce cadre peut donc s'adapter à de nouveaux types de données et à de nouvelles problématiques.

L'AFC est une méthode d'analyse de données centrée sur l'humain, définie comme un moyen de communication, d'exploration et de discussion [Pri06]. Pour citer Rudolf Wille [Wil02], "*le processus de création d'une structure conceptuelle encourage la découverte d'informations implicites et facilite la conversion des informations en connaissances*". L'AFC diffère donc des méthodes statistiques traditionnelles d'analyse d'informations. En structurant les configurations et les caractéristiques dans une structure unique mettant naturellement en évidence la variabilité, l'AFC apparaît comme un cadre propice à la rétro-ingénierie des LPLs.

1.2.2 Objectifs

Malgré ses propriétés naturelles d'analyse et de représentation de la variabilité, l'AFC a toujours été utilisée comme un moyen plutôt qu'un cadre pour la rétro-ingénierie. Nous pensons que le domaine de la rétro-ingénierie des LPLs pourrait bénéficier d'une étude plus détaillée de ce cadre mathématique et de ses propriétés, car il donne une représentation naturelle, réutilisable et extensible de la variabilité.

Notre objectif est donc **d'étudier les apports de l'utilisation de l'analyse formelle de concepts pour encadrer la rétro-ingénierie des LPLs**

Nous étudions pour cela les trois propriétés de l'AFC évoquées précédemment, que nous considérons comme trois sous-objectifs. Nous cherchons d'abord à définir les apports en terme de **représentation de la variabilité**. Nous avons vu que l'AFC est, entre autres, une méthode d'analyse de données et de représentation des connaissances, mais quelles sont les types d'informations qu'elle met en évidence et en quoi nous renseignent-elles du point de vue de la variabilité ? Le deuxième sous-objectif porte sur la **réutilisabilité** de

ce cadre de gestion d'informations : comment les structures de base de l'AFC peuvent-elles être réutilisées pour supporter les différentes opérations nécessaires à la gestion de la variabilité durant la migration ? Enfin, le troisième sous-objectif aborde les bénéfices de l'**extension** d'un tel cadre : peut-il être adapté à de nouveaux types de données ou de nouvelles problématiques ?

1.3 Contributions et organisation du manuscrit

1.3.1 Plan

Dans le Chapitre 2, nous étudions plus en détail les différents concepts évoqués dans la première partie de cette introduction. Plus spécifiquement, nous abordons les notions d'ingénierie des LPLs, de modélisation de la variabilité, de stratégies d'adoption et de rétro-ingénierie des LPLs. Ce chapitre a pour objectif d'établir fermement le contexte de nos travaux, afin de mieux comprendre et formuler les problématiques et les défis de la migration vers des LPLs à partir de variantes existantes.

Le Chapitre 3 introduit l'AFC. Nous en donnons les bases théoriques, puis nous étudions les informations sur la variabilité qui sont naturellement mises en évidence par ses structures conceptuelles. Nous montrons que les structures issues de l'AFC incluent naturellement la variabilité exprimée par les modèles de caractéristiques et qu'elles représentent des classes d'équivalence de ces modèles. Nous démontrons ainsi la pertinence de l'utilisation de ce cadre pour la structuration et la représentation de la variabilité en termes de caractéristiques d'un ensemble de variantes existantes, même si celles-ci n'ont pas de modèles de variabilité.

Dans le Chapitre 4, nous abordons le caractère réutilisable de l'AFC dans le contexte de la migration. Pour cela, nous proposons l'implémentation de trois opérations basées sur ce cadre et permettant de faciliter l'exploitation d'une collection de variantes logicielles : la synthèse de modèles de caractéristiques depuis des descriptions de variantes, la composition (union et intersection) de modèles de FPLs et la sélection de produits. Alors que la première opération cherche à obtenir un modèle représentant une FPL, l'implémentation des deux suivantes se base traditionnellement sur un tel modèle. Ainsi, nous montrons que l'AFC est un support à la rétro-ingénierie des modèles de caractéristiques et de par sa proximité avec ceux-ci, permet de supporter diverses opérations sur la variabilité, sans reposer sur les modèles de variabilité traditionnels.

L'extensibilité de l'AFC, toujours dans un contexte de migration, est étudiée au Chapitre 5. Nous montrons comment ce formalisme est capable de s'adapter et de se généraliser à d'autres informations concernant la variabilité. Pour cela, nous étudions les informations de variabilité étendue, exprimées par deux extensions des modèles de caractéristiques, consistant à leur rajouter des cardinalités et des attributs multivalués. Nous montrons comment l'AFC peut structurer des descriptions de variantes plus complexes et permet d'analyser et de structurer la variabilité étendue de la même manière que présenté au Chapitre 3.

Enfin, le Chapitre 6 conclut ce manuscrit et présente quelques perspectives.

1.3.2 Collaborations et inspirations

La grande majorité de ces travaux a été influencée par des échanges d'idées, de questions et d'autres travaux de la littérature que je souhaite citer ici.

Le Chapitre 2 est une étude des concepts principaux de l'ingénierie des lignes de produits logiciels, qui sont pour la plupart empruntés au livre *Software Product Line Engineering : Foundations, Principles and Techniques* [PBvdL05]. Nos travaux sont en partie motivés par les enquêtes de Berger et al. [BRN+13] et de Dubinsky [DRB+13] sur les pratiques liées à la gestion de la variabilité dans les entreprises.

Le Chapitre 3 est une étude de l'AFC pour la représentation de la variabilité. La Section 3.3 sur les informations de variabilité mises en évidence grâce à l'AFC regroupe les informations déjà identifiées dans la littérature par Loesch et Ploedereder [LP07] (i.e., les *core features* et les *dead features*) et propose une analyse bien plus poussée des structures conceptuelles, dans laquelle on peut certainement retrouver des idées de Godin et al. [GSG86]. La Section 3.4 sur l'extraction de variabilité depuis les structures conceptuelles propose une synthèse des articles sur le sujet [RPK11, AMHS+14, SSS17] et en extrait un processus unifié, complet et correct. La corrélation entre les modèles de caractéristiques et les structures de l'AFC présentée en Section 3.5 est originale, mais repose en partie sur les travaux de Czarnecki et al. [CW07] qui mettent en parallèle les modèles de caractéristiques et les formules propositionnelles. Une grande partie de l'étude de cette corrélation, notamment concernant les implications binaires, a été réalisée grâce à des échanges fructueux avec Karell Bertet et à ses connaissances sur les aspects structurels des treillis. Cela a donné lieu à une publication [CBHN16] puis à une extension soumise au journal *Discrete Applied Mathematics*.

Les contributions apportées dans le Chapitre 4, qui définit l'implémentation originale de trois opérations basée sur les structures de l'AFC, sont pour la plupart à partager avec des co-auteurs, incluant mes encadrantes de thèse. La Section 4.2 sur la synthèse de modèles de caractéristiques est très influencée par les travaux de Ryssel et al. [RPK11] et de Loesch et Ploedereder [LP07] sur la variabilité et les structures de l'AFC. Les ECFDs sont présentés dans un article [CHN17] et la synthèse de modèles est longuement présentée dans un article accepté dans le journal *Journal of Systems and Software*. La Section 4.3 s'inspire des définitions sur la composition de modèles de caractéristiques données par Acher et al. dans [ACLF10b]. Enfin la Section 4.4 sur la sélection de produits reprend la théorie exposée par Godin et al. dans [GSG86] sur la recherche exploratoire. Les expérimentations conduites à la fin des Sections 4.2 et 4.3 ont été réalisées grâce à l'implémentation des algorithmes et des structures dans l'outil `Talend` par André Miralles, chercheur à l'IRSTEA. Les travaux sur la composition de modèles ont été principalement élaborés sous sa tutelle et ne sont pas restreints aux LPLs ; il est à l'origine de la définition de l'intersection approximative. Ces travaux ont donné lieu à plusieurs publications [MHCN17, CHMN17] dont une à paraître dans les CCIS. La définition de l'algorithme de génération locale dans l'AOC-poset de la Section 4.4 est née d'une collaboration avec Alexandre Bazin et Giacomo Kahn, consacrés *clever young scientists* par Bernhard Ganter et respectivement post-doctorant et doctorant au LIMOS (Clermont-Ferrand). L'implémentation et les expérimentations sont cette fois de mon fait. Cette collaboration a donné lieu à une publication [BCK17]. Nous avons aussi étudié la génération locale dans les treillis connectés de l'analyse relationnelle de concepts mais ces travaux ne sont pas présentés dans ce manuscrit.

Enfin, le Chapitre 5 sur la gestion de la variabilité étendue prend ses racines dans des échanges d'idées avec Mathieu Acher lors de ses passages à Montpellier et d'un séminaire d'Amedeo Napoli sur les structures de patterns. La mise en place de ces méthodes s'inspire des travaux de Kaytoue-Uberall et al. sur les structures de patterns [KAMN10] et la représentation de données complexes pouvant être traitées avec l'AFC. L'extraction et la manipulation d'informations de variabilité étendue s'inspire des idées de Godin et Mili [GM93] sur l'utilisation de l'AFC pour introduire de nouvelles abstractions dans les diagrammes de classes UML. Ces travaux ont donné lieu à deux communications [CHN18, CHN18]

publiées et à un article soumis dans le journal *Journal of Systems and Software*.

1.3.3 Contributions

Dans cette section, nous résumons les contributions apportées dans chaque chapitre.

Chapitre 3

— analyse des informations contenues dans les structures conceptuelles du point de vue des configurations d'une FPL ;
— synthèse correcte et complète des méthodes d'extraction de variabilité basée sur l'AFC et correspondant à la sémantique logique des modèles de caractéristiques ;
— analyse des corrélations entre les structures de base de l'AFC et les modèles de caractéristiques booléens ;
— définition de l'ECFD, un modèle de classe d'équivalence des modèles de caractéristiques ayant la même sémantique de configurations mais des sémantiques ontologiques différentes.

Chapitre 4

— définition d'une méthode de synthèse semi-automatique de modèles de caractéristiques basée sur les ECFDs et le mapping entre les modèles de caractéristiques et les structures de base de l'AFC ;
— re-définition des opérations d'union stricte et d'intersection basées sur la sémantique de configurations des modèles de caractéristiques grâce aux structures de base de l'AFC et des ECFDs ;
— définition d'une opération d'intersection approximative basée sur les concepts introducteurs d'attributs des structures conceptuelles ;
— définition d'une méthode de sélection de produits par navigation conceptuelle dans les treillis de concepts et les AOC-posets ;
— définition d'un algorithme de génération locale du voisinage conceptuel dans les AOC-posets.

Chapitre 5

— structuration automatique des matrices de comparaison de produits (respectant un format défini) avec les structures de patterns ;
— extraction de relations logiques représentant la variabilité de modèles de caractéristiques étendus basée sur les structures de patterns ;
— proposition d'un ensemble d'heuristiques de réduction de la redondance dans les relations extraites avec les structures de patterns.

La contribution principale de ce manuscrit est d'avoir en partie unifié et rassemblé sous la coupe de l'AFC des travaux d'apparences très diverses, mais réalisant des traitements proches sur le plan théorique.

Chapitre 2

Sur la gestion de la variabilité d'une famille de produits logiciels

Préambule

L'objectif de ce chapitre est d'étudier différents concepts de l'ingénierie des lignes de produits logiciels, afin d'établir le contexte de nos travaux, puis d'en extraire et d'en comprendre les problématiques que nous allons traiter par la suite. Ce n'est donc pas un état de l'art détaillé ; les états de l'art seront donnés au fur et à mesure dans les autres chapitres. Nous présentons d'abord l'ingénierie des lignes de produits logiciels et nous abordons l'importance de la modélisation de la variabilité. Nous analysons ensuite les modèles de caractéristiques, qui sont les modèles de variabilité les plus utilisés, ainsi que leurs différentes sémantiques. Enfin, nous étudions les problématiques liées à la transition depuis des variantes logicielles développées sans effort de réutilisation vers des lignes de produits logiciels.

Sommaire

2.1 Introduction

Une famille de produits logiciels (FPL) désigne une collection de systèmes logiciels qui ont une cohérence générale dans leur objectif et qui partagent assez de similarités pour justifier leur gestion comme une seule entité (en considérant d'abord leur points communs) plutôt que de manière individuelle (en considérant d'abord leurs spécificités) [Par76]. L'ingénierie des lignes de produits logiciels [CN02, PBvdL05, VdL02] rassemble un ensemble de méthodes ayant pour objectif de produire une telle collection de systèmes logiciels similaires tout en réduisant leur coût de production, leur délai de commercialisation et en augmentant leur qualité ainsi que l'offre proposée [KBB+01]. C'est un paradigme de développement basé sur la dérivation d'une collection de variantes logicielles à partir d'un ensemble d'artefacts logiciels réutilisables, dont les assemblages possibles sont structurés par une architecture logicielle générique. L'expression "ligne de produits logiciels" (LPL) désigne à la fois l'architecture, les artefacts et les variantes qui peuvent en être dérivés. La construction *proactive* d'une telle structure requiert deux processus successifs, le premier ayant pour objectif de définir et de développer l'architecture et les artefacts (développement pour la réutilisation), et le second la mise en place de la dérivation de variantes depuis la plateforme définie précédemment (développement par la réutilisation) [CN02, PBvdL05].

Ces méthodologies ont été appliquées plusieurs fois et avec succès dans certaines entreprises, comme en témoigne le *product line hall of fame*[1] de la conférence *Software Product Line Conference* spécialisée sur les LPLs. Cependant, mettre en place une LPL ex nihilo en suivant la stratégie de construction proactive est un investissement à long terme, qui demande un temps et un coût non négligeables en amont, avant de produire les premiers produits logiciels commercialisables. A contrario, le développement de produits logiciels individuels est une stratégie qui s'avère rentable à plus court terme, mais la collection de variantes logicielles ainsi produite est plus difficile à maintenir et à faire évoluer par la suite. On note alors l'émergence d'une stratégie *extractive* combinant les avantages des deux précédentes, qui cherche à extraire une LPL en se basant sur les variantes existantes plutôt que de la construire ex nihilo [Kru01, Kru02]. Dans un premier temps, des variantes logicielles sont développées individuellement ou bien en utilisant des méthodes de réutilisation ad-hoc [DRB+13], assurant ainsi la rentabilité à court terme. Puis, dans un second temps, une LPL est construite à partir des variantes existantes [BRN+13], facilitant la gestion de la collection et assurant cette fois une rentabilité à plus long terme. Nos travaux se concentrent sur la mise en place d'une telle migration. La problématique principale étudiée dans ce manuscrit concerne donc la rétro-ingénierie des lignes de produits logiciels.

Les modèles de variabilité ont pour but de représenter les éléments communs, les éléments qui varient et comment ces éléments varient entre les produits de la LPL [KCH+90, MP07, CGR+12]. Ils sont établis au début du processus de développement pour la réutilisation et jouent un rôle central dans la gestion de la collection de variantes. Ils ont donc une importance toute particulière dans le processus de migration vers une LPL. Cependant, leur construction depuis des variantes existantes, qu'elle soit manuelle ou automatique, est une tâche ardue qui se heurte à différents problèmes [BABN16].

Dans ce qui suit, nous cherchons à établir le contexte de nos travaux afin de formuler nos objectifs plus en détails. Nous définissons d'abord l'ingénierie des lignes de produits logiciels en Section 2.2. Nous présentons les deux processus consécutifs permettant la construction et la mise en place proactive d'une LPL. Nous mettons en évidence l'importance de la représentation et de la modélisation de la variabilité des différents produits de la LPL, que nous étudions par la suite en Section 2.3. Le modèle de variabilité le plus populaire est appelé modèle de caractéristiques, que nous étudions en Section 2.4. Nous discutons

1. http ://splc.net/hall-of-fame/

de leurs différentes sémantiques et nous étudions les équivalences et les transitions entre ces différentes sémantiques. Enfin, nous présentons les différentes stratégies d'adoption des LPLs en Section 2.5. Nous nous concentrons sur la stratégie extractive, ainsi que sur les problématiques et les défis soulevés par son adoption. Nous concluons ce chapitre en Section 2.6 en mettant en évidence les questions de recherche que nous allons traiter dans le reste de ce manuscrit.

2.2 L'ingénierie des lignes de produits logiciels

Dans cette section, nous donnons une définition de l'ingénierie des lignes de produits logiciels, un paradigme de développement qui vise à produire des variantes logicielles similaires en réduisant leur coût et leur temps de développement grâce à la réutilisation. Nous faisons tout d'abord un point sur la terminologie, qui a évolué et qui varie encore aujourd'hui en fonction du continent, de la conférence ou des auteurs (Section 2.2.1). Puis, nous discutons du contexte dans lequel l'ingénierie des LPLs s'est développée (Section 2.2.2), avant d'en donner une définition (Section 2.2.3).

2.2.1 Un point sur la terminologie

David Parnas définit en 1976 une **famille de programmes** [Par76] comme un ensemble de systèmes logiciels assez similaires pour qu'il soit plus avantageux et pertinent de considérer d'abord leurs propriétés communes avant d'étudier celles qui sont spécifiques à chacun de ces produits. Il préconise alors de considérer et de développer l'ensemble de ces programmes comme une seule entité plutôt que de développer chacun d'entre eux séparément. La stratégie ainsi définie requiert de mettre en évidence les éléments communs à plusieurs systèmes logiciels afin de réutiliser les artefacts correspondants.

On retrouve par la suite le terme **famille de produits logiciels** [Bos00], qui décrit généralement un ensemble de systèmes logiciels similaires, accompagné d'une architecture générique autour de laquelle est organisé un ensemble d'artefacts communs utilisés pour produire de manière systématique ces produits logiciels. Ce terme est utilisé de manière analogue à celui de **ligne de produits logiciels**, défini par le *software engineering institute* comme un ensemble de systèmes logiciels qui partagent un ensemble de caractéristiques communes, satisfaisant les besoins spécifiques d'un segment de marché et qui sont développés selon un cadre déterminé à partir d'un ensemble d'artefacts communs. David Parnas différencie cependant une famille de produits logiciels d'une ligne de produits logiciels, le premier terme faisant référence à un concept d'ingénierie et le second à un concept marketing et commercial.

D'autre part, le terme de **famille de produits logiciels** fait souvent référence à un ensemble de produits similaires, sans architecture générique ni artefacts logiciels réutilisables, de manière assez similaire à la famille de programmes. Pour qu'il n'y ait pas d'ambiguïté par la suite, nous parlerons de famille de produits logiciels (FPL) pour désigner des produits logiciels ayant des éléments et propriétés communs mais qui n'ont pas été développés avec des méthodologies axées sur la réutilisation. Nous utiliserons le terme ligne de produits logiciels (LPL) pour faire référence à une FPL décrite par une architecture générique définissant les éléments communs et spécifiques à ses différents systèmes et autour de laquelle s'articule un ensemble d'artefacts logiciels réutilisables. Ce dernier terme est en effet plus utilisé dans la littérature actuelle et il est plus cohérent avec celui d'ingénierie des lignes de produits logiciels.

2.2.2 Contexte

Supposons qu'un certain nombre d'utilisateurs souhaitent utiliser une même catégorie de logiciels pour réaliser une certaine tâche, mais que les fonctionnalités utilisées par chaque utilisateur puissent être légèrement différentes. Dans ces cas, deux stratégies de développement prédominent : le développement de logiciels standards et le développement de logiciels individuels [PBvdL05, HP03].

Le développement d'un **logiciel standard** consiste à regrouper dans une même distribution du logiciel toutes les fonctionnalités pouvant intéresser les utilisateurs d'un segment de marché. Cette distribution est ensuite proposée à tous ces utilisateurs. Les exigences auxquelles doit répondre un tel logiciel sont identifiées lors d'analyses de marché préalables et doivent prendre en compte un périmètre assez étendu afin de répondre aux attentes du plus grand nombre d'utilisateurs possible. Cette stratégie demande un temps et un coût de production important, mais une seule fois. Cependant, l'utilisateur acquiert un logiciel onéreux et de taille importante alors que potentiellement, seule une partie des fonctionnalités offertes lui sera utile.

Exemple. La partie gauche de la Figure 2.1 illustre cette stratégie. On peut voir un système logiciel implémentant quatre fonctionnalités différentes indiquées par 4 couleurs : certains utilisateurs n'utilisent qu'une fonctionnalité (e.g., les utilisateurs 1 et 3), d'autres en utilisent plusieurs (e.g., l'utilisateur 2) et certains utilisent les mêmes fonctionnalités du logiciel standard (e.g., les utilisateurs 1 et 4). Notons que la plupart des utilisateurs représentés n'utilisent qu'un quart du logiciel qu'ils ont acheté.

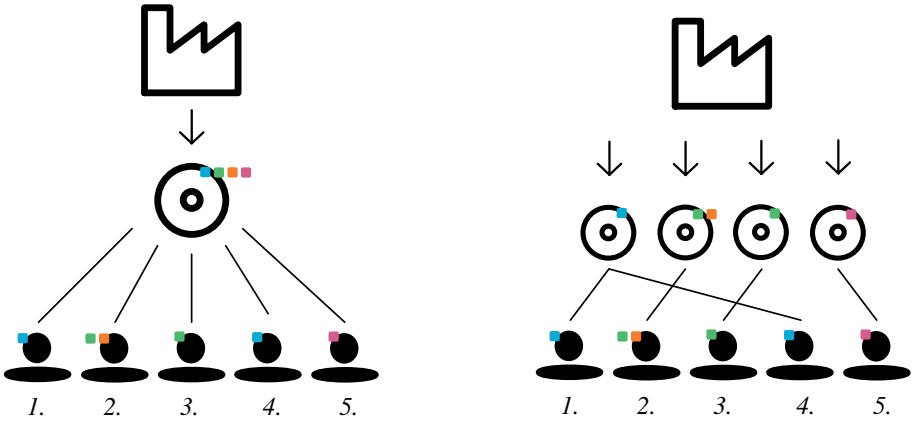


FIGURE 2.1 – (gauche) développement et utilisateurs de logiciel standard ; (droite) développement et utilisateurs de logiciels individuels ; les couleurs indiquent les différents artefacts logiciels

Le développement d'un **logiciel individuel** propose de produire un logiciel spécifique pour chaque utilisateur. Cette fois, les exigences du système logiciel à produire sont explicitées par l'utilisateur, qui devrait obtenir un produit qui correspond à ses attentes et ne contenant que les fonctionnalités qui lui sont nécessaires. Cette stratégie demande un temps et un coût de développement moins importants que les logiciels standards, mais qui devront être réitérés pour chaque logiciel.

Exemple. Cette stratégie est illustrée dans la partie droite de la Figure 2.1. On y voit des logiciels de taille plus réduite que la version standard, implémentant les fonctionnalités correspondant aux exigences des utilisateurs. On peut observer que certains utilisateurs peuvent avoir les mêmes exigences et bénéficier d'une même variante (e.g., utilisateurs 1 et 4) et certains systèmes produits peuvent avoir des fonctionnalités communes bien qu'ils soient différents (e.g., systèmes logiciels des utilisateurs 2 et 3).

Une stratégie intéressante est d'allier les avantages des deux précédentes : la capacité d'adapter un système logiciel aux exigences d'un utilisateur (avantage du développement de logiciels individuels) et la capacité de satisfaire à moindre coût le plus grand nombre d'utilisateurs possible (avantage du développement de logiciels standards), grâce au "développement en une fois" couplé au large périmètre d'utilisation.

Le premier défi est d'être capable de prendre en compte les exigences d'un grand nombre d'utilisateurs potentiels. On cherche pour cela à produire des logiciels plus spécifiques et adaptés que les logiciels standards, mais de manière plus rapide et moins chère que les logiciels individuels : c'est la **personnalisation de masse**.

Définition 2.1 (Personnalisation de masse)**.** *La personnalisation de masse est la production à grande échelle de produits adaptés aux exigences de clients individuels.*

De plus, nous pouvons soulever deux observations en étudiant les utilisateurs 2 et 3 du schéma à droite de la Figure 2.1 : a) les développeurs redéfinissent régulièrement les mêmes fonctionnalités à travers différents logiciels indépendants, et b) certains logiciels correspondent à une spécialisation d'un logiciel préexistant. Une culture de la **réutilisation systématique** émerge de ces deux constatations.

Définition 2.2 (Réutilisation systématique)**.** *Dans le domaine du développement logiciel, la réutilisation systématique est une approche organisationnelle du développement qui consiste à définir des artefacts réutilisables et flexibles, qui seront ensuite utilisés à travers plusieurs systèmes logiciels d'une manière prescrite afin de minimiser le temps et le coût de production de ces logiciels.*

La personnalisation de masse couplée à la réutilisation systématique font naître une stratégie de développement laissant à l'utilisateur la possibilité de définir ses exigences et de faciliter et d'accélérer le développement du logiciel adéquat en sélectionnant les artefacts correspondants afin de composer efficacement le logiciel final. C'est sur ces deux principes que se base **l'ingénierie des lignes de produits logiciels**.

2.2.3 Définitions

L'**ingénierie des lignes de produits logiciels** [PBvdL05] est une approche basée sur la réutilisation systématique et la personnalisation de masse visant à réduire le temps et le coût de développement d'un ensemble de systèmes logiciels similaires. Ce paradigme cherche à développer une FPL comme une seule entité (en privilégiant les éléments communs des variantes logicielles), plutôt que comme une somme de logiciels similaires individuels (en privilégiant leurs éléments spécifiques). Le cœur de cette approche est basé sur le développement d'une **architecture logicielle générique**, qu'il est possible de combiner avec divers **artefacts logiciels réutilisables** en fonction d'un ensemble d'exigences. Ainsi, plusieurs systèmes logiciels différents bien que fortement similaires, appelés **variantes logicielles**, peuvent être dérivés de cette architecture commune couplée à des artefacts variables. L'architecture générique regroupe donc l'ensemble des fonctionnalités communes à toutes les futures variantes et présente en plus des **points de variation** correspondant à des fonctionnalités optionnelles qu'un utilisateur peut choisir d'ajouter ou non à la variante finale. L'architecture générique, les artefacts réutilisables et l'ensemble des variantes logicielles qui peuvent en être dérivées forment ce que l'on appelle une **ligne de produits logiciels** (LPL).

La Figure 2.2 illustre le fonctionnement de cette stratégie de développement. La réutilisation systématique se met en place à travers l'architecture générique, qui est un moyen de structurer et d'encadrer la réutilisation des artefacts. La personnalisation de masse est

rendue possible grâce à la composition de l'architecture en fonction des points de variation et des exigences de l'utilisateur. On peut donc faire la distinction entre deux étapes : le développement pour la réutilisation et le développement par la réutilisation.

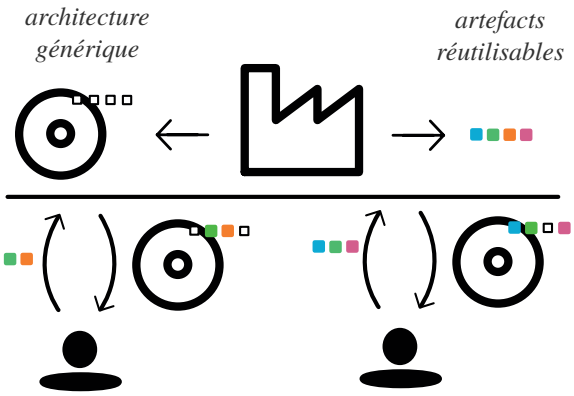


FIGURE 2.2 – Les LPLs : une approche basée sur la réutilisation systématique et la personnalisation de masse ; les couleurs indiquent les différents artefacts logiciels

Processus et méthodologie

L'ingénierie des LPLs est un paradigme de développement qui se décompose en deux processus [VdL02, PBvdL05] : l'ingénierie du domaine, suivie de l'ingénierie d'application.

L'**ingénierie du domaine** est le processus durant lequel sont caractérisés puis développés les points communs et les variations de la LPL, i.e., l'architecture générique et ses points de variation, ainsi que les artefacts réutilisables. L'objectif est donc le **développement pour la réutilisation**. C'est durant ce processus qu'est déterminé le périmètre de la LPL, c'est-à-dire les variantes logicielles qui vont pouvoir être dérivées par la LPL. Ce processus se décompose en trois phases, illustrées en Figure 2.3

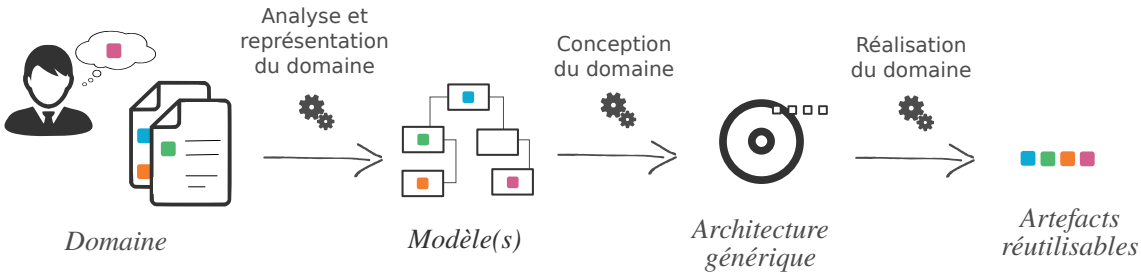


FIGURE 2.3 – Les trois phases de l'ingénierie du domaine : le développement pour la réutilisation

L'**analyse et la représentation du domaine** est la première étape, qui a pour but de mettre en évidence et de documenter les éléments communs et spécifiques aux différentes variantes logicielles de la LPL. Elle est réalisée par des experts du domaine et des concepteurs qui doivent définir les exigences et le périmètre de la LPL. Il en résulte des modèles représentant la **variabilité** à un haut niveau d'abstraction de la LPL et qui cherchent à répondre à ces questions : Quels sont les éléments communs ? Quels sont les éléments variables, qui sont spécifiques à certaines variantes ? Comment ces éléments varient-ils d'une variante à une autre ? Ces modèles documentent entre autres les contraintes métiers que devront respecter l'implémentation de l'architecture et des artefacts réutilisables.

La **conception du domaine** se base sur les modèles de variabilité établis lors de l'étape précédente afin de concevoir et d'implémenter l'architecture générique et les points de variation. Durant cette étape, les modèles de variabilité qui avaient jusqu'ici un fort niveau de granularité et représentaient des fonctionnalités ayant un haut niveau d'abstraction, sont raffinés afin de définir des niveaux de granularité plus bas, plus techniques, moins abstraits.

La **réalisation du domaine** est la dernière étape de ce processus, consistant à implémenter les artefacts réutilisables de la LPL, correspondant aux fonctionnalités de haut niveau visibles par l'utilisateur, qui vont pouvoir être combinées à l'architecture définie précédemment. Cette étape se base donc sur l'architecture générique développée précédemment, ainsi que sur la spécification des artefacts réutilisables qui a été définie lors de la première étape, puis raffinée lors de la seconde.

L'**ingénierie d'application** correspond cette fois au **développement par la réutilisation**. Son objectif est d'exploiter la variabilité mise en place lors de l'ingénierie du domaine pour faciliter le développement des variantes logicielles de la LPL. C'est un processus qui démarre depuis un ensemble d'exigences d'un client et qui produit une variante logicielle qui correspond aux exigences données. On distingue deux phases, qui sont illustrées en Figure 2.4.

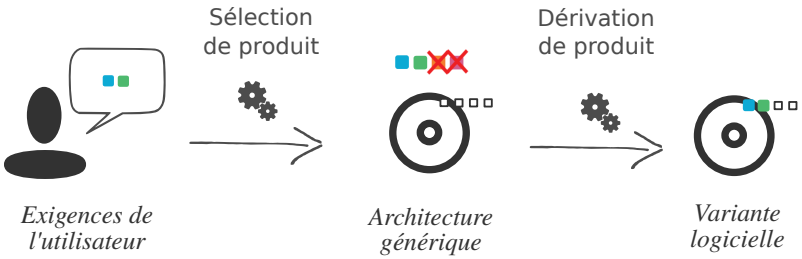


FIGURE 2.4 – Les deux étapes de l'ingénierie d'application : le développement par la réutilisation

La **sélection de produit** est l'étape durant laquelle l'utilisateur va spécifier ses exigences concernant la variante logicielle finale. Pour cela, les fonctionnalités sélectionnées et les artefacts réutilisables qui leur sont associés doivent respecter les contraintes métiers relatives au domaine et à l'implémentation de l'architecture, afin de garantir la sélection d'une variante opérationnelle.

La **dérivation du produit** est la dernière étape durant laquelle est assemblée la variante logicielle finale, en combinant à l'architecture générique les artefacts réutilisables correspondant à la sélection de l'utilisateur.

En résumé (2.2) :

- Une **famille de produits logiciels** désigne un ensemble de systèmes logiciels similaires.
- Une **ligne de produits logiciels** fait référence à une famille de produits logiciels dont la production est encadrée à travers une architecture générique et un ensemble d'artefacts logiciels communs et réutilisables.
- L'ingénierie des LPLs est un paradigme de développement basé sur la **réutilisation systématique** et la **personnalisation de masse**.

— Ce paradigme se décompose en deux processus : l'**ingénierie du domaine**, durant laquelle les éléments communs et les variations au sein de la famille de produits sont caractérisés et implémentés et l'**ingénierie d'application**, durant laquelle la variabilité mise en place est exploitée pour dériver des variantes logicielles opérationnelles.

Les processus de mise en place d'une LPL se basent donc principalement sur la représentation et la description de sa **variabilité**. Dans la section suivante, nous discutons des approches de modélisation de cette variabilité.

2.3 La modélisation de la variabilité

La modélisation de la variabilité est une tâche centrale de l'ingénierie des LPLs. Elle consiste à représenter les éléments communs, les éléments variables et comment ces éléments varient dans les produits d'une LPL. Cette représentation doit être synthétique et compréhensible par tous les intervenants (les concepteurs, les développeurs, les utilisateurs finaux, etc.).

Un modèle de variabilité permet de faciliter la gestion d'une LPL en offrant un support pour des tâches telles que la représentation, la manipulation, la recherche d'information, l'évolution ou encore la maintenance d'un ensemble de variantes logicielles. Plusieurs approches de modélisation de la variabilité coexistent : chacune a son propre formalisme motivé par ses propres objectifs de modélisation. Les deux approches les plus répandues sont dites *orientée caractéristiques* et *orientée décisions* [CGR+12]. Nous les détaillons ci-après.

2.3.1 Approche orientée caractéristiques

Les modèles orientés caractéristiques ont pour objectif de documenter la variabilité en termes de fonctionnalités ou de comportements des variantes logicielles [KCH+90, KLD02, VGBS01]. On appelle *caractéristiques* des éléments permettant de spécifier et de distinguer les produits d'une LPL. Le livre *Feature-Oriented Software Product Line* [ABKS13, p.18] regroupe 10 définitions différentes du terme "caractéristique" pour donner la suivante :

Définition 2.3 (Caractéristique)**.** *Comportement d'un système logiciel visible par un utilisateur. Les caractéristiques sont utilisées dans l'ingénierie des LPLs pour spécifier et communiquer les points communs et les différences des produits aux différents intervenants et pour guider la structuration, la réutilisation et les variations à travers chaque étape du cycle de vie d'un système logiciel.*

Les caractéristiques servent à tracer un lien entre les exigences spécifiées par l'utilisateur et les artefacts réutilisables de la LPL. Elles représentent généralement des fonctionnalités de haut niveau, qui sont compréhensibles par tous les intervenants (e.g., utilisateurs, concepteurs). Une caractéristique est implémentée par un ou plusieurs artefacts de la LPL, qui sont de plus bas niveau. À une exigence peut être associé un ensemble de caractéristiques.

Les **modèles de caractéristiques** [KCH+90, KLD02] sont les plus utilisés pour représenter la variabilité en termes de caractéristiques. Ces modèles représentent et organisent graphiquement les différentes caractéristiques dans une hiérarchie de spécialisation et définissent des relations (aussi appelées contraintes ou dépendances) entre ces caractéristiques. Ces relations sont exprimées à travers des éléments graphiques ajoutés à la hiérarchie, ou

par des contraintes textuelles additionnelles. Le formalisme des modèles de caractéristiques sera discuté plus en détails dans la Section 2.4, qui leur est consacrée. Les relations entre les caractéristiques contraignent la façon dont elles peuvent être combinées dans un produit de la LPL ; en d'autres termes, ces modèles délimitent le périmètre de la LPL en représentant de manière compacte et graphique un ensemble de combinaisons autorisées.

Exemple. La Figure 2.5 montre un exemple de modèle de caractéristiques représentant une LPL sur des applications de e-commerce. On peut y lire par exemple que :

— tous les `e_commerce` possèdent obligatoirement une fonctionnalité `catalog` ;
— tous les `e_commerce` peuvent éventuellement posséder une fonctionnalité `basket` ;
— si un `e_commerce` possède la fonctionnalité `basket`, il doit aussi avoir la fonctionnalité `payment_method`.

2.3.2 Approche orientée décisions

Les **modèles de décisions** sont des représentations textuelles qui ont pour objectif de documenter chaque décision qu'un utilisateur peut prendre durant la phase de sélection de produit de l'ingénierie d'application [SJ04, CGR+12]. Ces modèles sont généralement définis sous la forme d'une table, dans laquelle chaque ligne décrit une décision sous la forme d'une question accompagnée du type de réponse attendue ou bien de ses valeurs possibles. Chacune de ces questions concerne une fonctionnalité de la LPL correspondant à un ou plusieurs artefacts réutilisables. Les modèles de décisions peuvent aussi présenter des contraintes sur la forme de la réponse attendue (comme par exemple sa cardinalité). Des conditions définissant les cas dans lesquels la question doit être présentée à l'utilisateur peuvent aussi être ajoutées.

Exemple. La Table 2.1 présente un modèle de décisions sur des applications de e-commerce, inspiré de la Figure 2.5. La première ligne décrit la décision nommée `basket`, concernant la présence d'un support pour l'achat en ligne. La réponse attendue doit être *true* ou bien *false*. Si la décision nommée `payment method` a déjà été prise et que la réponse donnée est *true*, alors la réponse pour la décision `basket` est forcément *true*.

TABLE 2.1 – Exemple de modèle de décisions pour des applications de e-commerce

id de la décision	**description**	**type**	**valeurs**	**cardinalité, contraintes**	**visibilité**
`basket`	support pour l'achat en ligne ?	Bool	*true*/*false*	si `payment method` alors *true*	
`online payment`	support pour payer en ligne ?	Bool	*true*/*false*	si `basket` alors *true*	
`payment method`	quelles méthodes de paiement en ligne ?	Enum	*{check, card}*	[1..2]	si `online payment`
`catalog`	quelle interface de catalogue ?	Enum	*{grid, list}*	[1..1]	
`quick purchase`	support pour achat rapide ?	Bool	*true*/*false*	si *true* alors `payment method` $\not\subseteq$ `check`	si `basket`

2.3.3 Comparaison

Alors que les modèles de caractéristiques se concentrent sur la représentation du domaine à travers la description des caractéristiques distinguables et de leurs interactions, les

modèles de décisions cherchent à représenter le processus de sélection de produit en cataloguant les différentes décisions que peut faire un utilisateur lors du choix d'une variante de la LPL. Ainsi, les modèles de décisions guident l'utilisateur dans la sélection d'une variante, mais ne donnent pas nécessairement une vue d'ensemble complète de la LPL. En effet, même si certaines parties de la LPL peuvent être identifiées avec un modèle de décisions (e.g., *card* et *check* sont deux méthodes de paiement possibles, la fonctionnalité *basket* est optionnelle), un tel modèle ne couvre pas à coup sûr l'ensemble des informations concernant le domaine, contrairement à un modèle de caractéristiques. Par exemple, on peut voir dans le modèle de caractéristiques de la Figure 2.5 que `catalog` est une caractéristique raffinant `e_commerce`, mais qui doit être présente dans toutes les variantes de la LPL (contrainte indiquée par le cercle noir, forçant sa sélection). Cependant, cette information n'apparaît pas clairement dans le modèle de décisions de la Table 2.1. Parce que la fonctionnalité `catalog` est obligatoire, elle ne représente aucune décision de l'utilisateur ; seule la représentation de ce catalogue est soumise au choix de l'utilisateur. En conclusion, on peut dire que l'approche orientée décisions est plus utile du point de vue d'un utilisateur, alors que l'approche orientée caractéristiques sert les concepteurs tout au long du cycle de vie de la LPL.

En résumé (2.3) :

- La **modélisation de la variabilité** est une tâche centrale de l'ingénierie des LPLs qui facilite la gestion d'un ensemble de variantes logicielles.
- Il existe deux principales approches de modélisation : une **orientée décisions** qui se concentre sur la variabilité du point de vue d'un utilisateur et une **orientée caractéristiques** qui se concentre sur la variabilité du domaine étudié, plus utile pour les concepteurs de la LPL.
- Les **caractéristiques** servent à distinguer et comparer les différentes variantes et permettent de faire un pont entre les **exigences** de l'utilisateur et les **artefacts** de la LPL.

Dans ce manuscrit, nous nous concentrons sur l'approche orientée caractéristiques, qui est la plus courante et la plus pertinente du point de vue d'un concepteur. Dans ce qui suit, nous présentons plus en détails les modèles de caractéristiques, qui sont le standard de facto de cette approche.

2.4 Les modèles de caractéristiques

Les modèles de caractéristiques [KCH+90] (*feature models* en anglais, abrégé en FMs) sont des modèles de variabilité. Leur but est donc de caractériser quels éléments d'une LPL sont communs à tous les produits, lesquels peuvent varier d'un produit à un autre et comment ces éléments peuvent varier. En d'autres termes, ils modélisent les exigences communes et variables dans la LPL. Les FMs sont utilisés comme support pour plusieurs tâches de l'ingénierie des LPLs, principalement pour la représentation d'informations [CKK06], mais aussi pour définir le périmètre de la LPL, son évolution et sa maintenance [LP07], la réalisation d'opérations de conception [ACLF10b, ACLF10a] ou bien la dérivation de produits [JSvGB04]. Dans cette section, nous allons tout d'abord définir les FMs et illustrer ces définitions sur un exemple. Puis, nous allons étudier leurs différentes sémantiques et terminer par une analyse approfondie de l'équivalence entre ces sémantiques.

2.4.1 Définitions

Les FMs sont une famille de langages de notations graphiques. Ils représentent un ensemble de caractéristiques ainsi que des relations entre ces caractéristiques sous la forme d'un diagramme. Un exemple de FM dans le langage proposé par FODA [KCH+90], nommé FM_{ec}, est donné en Figure 2.5 ; il représente une LPL d'applications de commerce en ligne.

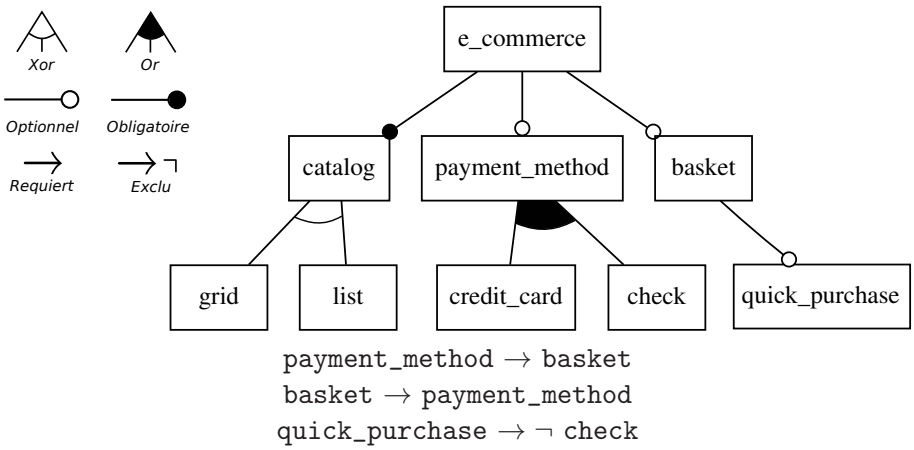


FIGURE 2.5 – Exemple de FM pour des applications de e-commerce (FM_{ec})

Chaque caractéristique est représentée par son nom (unique) dans un encadré. Ces caractéristiques sont organisées hiérarchiquement dans un arbre, appelé *arbre des caractéristiques*. Cet arbre sert de guide lors du processus de sélection des caractéristiques qui vont composer le produit final. La caractéristique racine est le point d'entrée de ce processus : elle représente une abstraction de toutes les caractéristiques de l'ensemble considéré et représente généralement le nom de la LPL modélisée. Dans notre exemple, la racine porte le nom de `e_commerce` et sert d'abstraction à toutes les fonctionnalités que l'on trouve dans des applications de commerce en ligne.

L'arbre des caractéristiques représente des relations *mère-fille* entre les caractéristiques et leurs sous-caractéristiques. Ces relations peuvent exprimer la spécialisation (les caractéristiques filles raffinent la caractéristique mère), mais aussi la composition (la caractéristique mère se compose de ses caractéristiques filles), ou encore l'utilisation (la caractéristique mère emploie ses caractéristiques filles). De ce fait, si l'on choisit de sélectionner une caractéristique quelconque de l'arbre, alors les caractéristiques plus générales qui en représentent les abstractions, i.e., celles situées entre la caractéristique sélectionnée et la racine, devront aussi être sélectionnées. Par exemple, si un utilisateur souhaite sélectionner la caractéristique `check`, alors `payment_method` ainsi que `e_commerce` devront aussi être sélectionnées.

De plus, les branches de l'arbre sont décorées pour exprimer des contraintes sur la sélection des caractéristiques filles lorsque la caractéristique mère est sélectionnée. Un disque noir indique que lorsque la caractéristique mère est sélectionnée, la caractéristique fille doit être obligatoirement sélectionnée aussi (*relation obligatoire*). C'est le cas entre `e_commerce` et `catalog` : `e_commerce` étant la racine, `catalog` devra donc toujours être sélectionnée. Cela signifie que toutes les applications de commerce en ligne possèdent un catalogue. Un disque blanc n'impose aucune contrainte : si la caractéristique mère est sélectionnée, la caractéristique fille peut l'être, ou non (*relation optionnelle*). C'est le cas pour `basket` et `quick_purchase` : si l'utilisateur choisit d'ajouter une fonctionnalité de gestion de panier dans son site de commerce en ligne, il peut aussi choisir d'y ajouter la fonctionnalité d'achat rapide ou non. Il est possible de grouper les caractéristiques filles pour exprimer

des contraintes sur le nombre de caractéristiques du groupe qui peuvent ou doivent être sélectionnées. Un groupe est représenté par un arc reliant les caractéristiques qui y sont incluses. Si l'arc est rempli en noir, il exprime un *groupe or* : si la caractéristique mère est sélectionnée, au moins une des caractéristiques filles du groupe doit être sélectionnée. Les caractéristiques `credit_card` et `check` forment un groupe *or* sous `payment_method` : l'utilisateur ayant choisi la fonctionnalité de méthode de paiement doit donc la raffiner en précisant s'il souhaite que son commerce en ligne propose un paiement par carte bancaire, par chèque, ou bien les deux. Si l'arc n'est pas rempli, il exprime alors un *groupe xor* : la sélection de la caractéristique mère impose alors la sélection d'exactement une des caractéristiques filles du groupe. Dans notre exemple, `catalog` possède un groupe *xor* composé de `grid` et `list` : l'utilisateur doit choisir le type d'affichage de son catalogue (par grille ou par liste), mais il ne peut pas sélectionner les deux.

D'autres contraintes qui ne peuvent pas être exprimées dans l'arbre peuvent être ajoutées au modèle : on les appelle les *contraintes transverses* et elles représentent généralement des implications binaires ou des exclusions mutuelles (aussi appelées *mutex*) sous forme propositionnelle. Par exemple, `payment_method` $\rightarrow$ `basket` est une implication binaire transverse qui indique que si l'utilisateur sélectionne la fonctionnalité méthode de paiement, son application devra aussi posséder une gestion de panier ; `quick_purchase` $\rightarrow \neg$ `check` est un mutex transverse interdisant à l'utilisateur de sélectionner la méthode de paiement par chèque si la fonctionnalité d'achat rapide est sélectionnée. On peut cependant trouver d'autres types de contraintes telles que les *soft constraints* [BM11], qui sont moins strictes que les contraintes précédentes (par exemple, "la présence de A encourage la présence de B"). Ces contraintes ne sont pas étudiées ici.

Le FM de la Figure 2.5 décrit une famille d'applications de commerce en ligne qui possèdent obligatoirement un catalogue ; ce catalogue est soit affiché sous forme de grille, soit sous forme de liste, mais pas les deux ; ces applications peuvent proposer de remplir un panier, ou non ; elles peuvent implémenter des méthodes de paiement par carte de crédit, par chèque, ou bien les deux ; si la fonctionnalité de gestion de panier est présente, au moins une méthode de paiement doit être implémentée et inversement ; une application peut proposer une fonctionnalité d'achat rapide si elle possède une gestion de panier ; l'achat rapide est incompatible avec l'implémentation de la méthode de paiement par chèque.

She et al. [SRA+14] formalisent la définition d'un FM comme suit :

Définition 2.4 (Modèle de caractéristiques)**.** *Un FM est un tuple* $FD = (F, E, (E_m, E_i, E_x), (G_o, G_x, G_m))$*, où* F *est un ensemble fini de caractéristiques ;* $E \subseteq F \times F$ *est un ensemble d'arcs de type "mère-fille" ;* $E_m \subseteq E$ *est un ensemble d'arcs de type "obligatoire" ;* $E_i \subseteq F \times F$ *est un ensemble d'implications binaires transverses avec* $E_i \cap E = \varnothing$ *;* $E_x \subseteq F \times F$ *est un ensemble d'exclusions mutuelles transverses avec* $E_x \cap E = \varnothing$ *; les ensembles* G_o, G_x *et* G_m *contiennent des sous-ensembles de* E *disjoints, participant respectivement à des groupes or, xor et mutex : chaque sous-ensemble de* G_o, G_x *ou* G_m *est disjoint de tous les autres sous-ensembles appartenant à ces ensembles.*

Les contraintes suivantes garantissent que le modèle est bien formé :

— (F, E) *est un arbre enraciné connectant toutes les caractéristiques de* F *;*

— *toutes les arêtes d'un groupe partagent le même parent.*

Notons que dans leur article, les auteurs rajoutent des groupes appelés groupes *mutex* qui sont des groupes dans lesquels au maximum une caractéristique peut être sélectionnée, sans obligation d'en sélectionner au moins une. Les groupes *mutex* sont différents des mutex transverses, car ces derniers ne sont pas enracinés. Ces groupes ne sont pas présents dans la notation FODA et ne seront pas pris en compte dans ce manuscrit. Ils peuvent cependant être représentés par une combinaison de relations optionnelles et d'exclusions

mutuelles transverses : les ignorer ne réduit donc pas les contraintes exprimables par le modèle. De plus, les auteurs font la distinction entre ce qu'ils appellent un diagramme de caractéristiques et un modèle de caractéristiques. Ils désignent par l'expression "diagramme de caractéristiques" un FM classique tel que défini précédemment (Définition 2.4) et par "modèle de caractéristiques" un diagramme de caractéristiques accompagné d'une formule propositionnelle quelconque ajoutant des contraintes aux caractéristiques de F.

En contraignant la sélection, un FM décrit un ensemble des combinaisons valides de caractéristiques, qui correspond aux variantes logicielles fonctionnelles de la LPL. Les contraintes du modèle représentent donc les contraintes métiers que doivent respecter les fonctionnalités implémentées dans une variante logicielle. Les FMs peuvent convier d'autres types d'informations, que nous détaillons dans la sous-section suivante.

2.4.2 Sémantiques des FMs

On distingue trois types d'informations (i.e., sémantiques) donnés par un FM.

Sémantique de configurations Le terme *configuration* définit un sous-ensemble des caractéristiques présentées dans le modèle. On dit qu'une configuration est valide si elle respecte toutes les contraintes exprimées par le FM. Le FM représente donc un ensemble de configurations valides ; il définit de cette manière le périmètre d'une LPL [SLB+11], i.e., l'ensemble des produits qui peuvent être dérivés. L'ensemble des configurations valides définies par un FM correspond à sa sémantique de configuration. Pour un FM $\mathcal{F}$, on la note $[\![\mathcal{F}]\!]$. Ces informations sont utiles dans le contexte de la configuration et de la dérivation de variantes valides.

Exemple. Le FM FM_{ec} présenté en Figure 2.5 définit les 10 configurations suivantes :

$[\![FM_{ec}]\!] =$
{{e_commerce, catalog, grid},
{e_commerce, catalog, list},
{e_commerce, catalog, grid, payment_method, check, basket},
{e_commerce, catalog, list, payment_method, check, basket},
{e_commerce, catalog, grid, payment_method, credit_card, basket},
{e_commerce, catalog, list, payment_method, credit_card, basket},
{e_commerce, catalog, grid, payment_method, credit_card, check, basket},
{e_commerce, catalog, list, payment_method, credit_card, check, basket},
{e_commerce, catalog, grid, payment_method, credit_card, basket,
quick_purchase},
{e_commerce, catalog, list, payment_method, credit_card, basket,
quick_purchase}}

Sémantique ontologique Gruber [Gru95] définit simplement une ontologie comme étant un ensemble de spécifications explicites d'une conceptualisation. À notre connaissance, le parallèle entre les FMs et les ontologies fut réalisé par Czarnecki et al. [CKK06]. Plus tard, She et al. utiliseront l'expression "sémantique ontologique d'un FM" pour indiquer le sens donné par les caractéristiques et l'arbre des caractéristiques [SLB+11]. En effet, les caractéristiques représentent des fonctionnalités à haut niveau de granularité et peuvent être perçues comme des concepts abstraits ou concrets du domaine de la LPL modélisée. De plus, les types de relations reliant ces caractéristiques (e.g., relations mère-fille, sélection obligatoire, groupes) ont aussi un sens ontologique : ces relations apportent des informations concernant les interactions entre les caractéristiques représentant le domaine. Cependant, bien que ces modèles permettent de décrire des concepts, leur expressivité reste

très limitée comparée aux langages de descriptions utilisés par les ontologies traditionnelles [CKK06].

Exemple. Le FM de la Figure 2.5 indique que :
— `e_commerce` désigne la LPL modélisée ;
— `catalog`, `payment_method` et `basket` sont des caractéristiques plus spécifiques que `e_commerce` et expriment des relations de composition ;
— `catalog` est obligatoire dans toutes les configurations ;
— `grid` et `list` spécialisent `catalog` : ce sont deux façons de représenter un catalogue ;
— `credit_card` et `check` spécialisent `payment_method` ;
— `quick_purchase` raffine la gestion du panier : c'est une sous-caractéristique optionnelle.

La sémantique ontologique est importante pour comprendre, maintenir et exploiter un modèle, que ce soit de manière automatique ou bien manuelle. Becan et al. [BABN16] illustrent la nécessité d'avoir des modèles ayant une sémantique ontologique cohérente, en construisant automatiquement des configurateurs qui s'avèrent être absurdes car dérivés de modèles incohérents.

Sémantique logique À notre connaissance, le lien entre les FMs et la logique du premier ordre doit sa paternité à Mike Mannion [Man02]. Les contraintes exprimées par les relations du modèle ont depuis été transcrites en logique propositionnelle : chaque caractéristique correspond à une variable propositionnelle et chaque contrainte est définie en utilisant les connecteurs propositionnels ($\wedge, \vee, \rightarrow, \leftrightarrow, \rightarrow \neg$ et $\oplus$). On appelle sémantique logique d'un FM sa représentation sous la forme d'un ensemble de formules propositionnelles. De nombreux *mappings* équivalents (i.e., associant à un même FM des formules propositionnelles équivalentes) sont présents dans la littérature [Bat05, CW07, PSA+12, SRA+14] ; la Table 2.2 en est une représentation.

De nombreux travaux reposent sur la sémantique logique des FMs, notamment pour faire du raisonnement automatique [BSRC10]. Les exemples d'opérations pouvant être automatisées comprennent (mais ne sont pas limitées à) : déterminer si un modèle représente au moins une configuration valide [TBK09, MWC09], déterminer si une configuration donnée est valide, lister l'ensemble des configurations valides, compter le nombre de configurations [BTRC05], détecter les caractéristiques non utilisées [TBD+08].

Il est donc utile de pouvoir passer de la représentation d'une sémantique à une autre, en fonction du contexte et des tâches à effectuer.

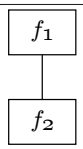
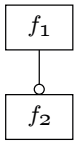
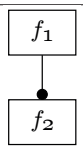
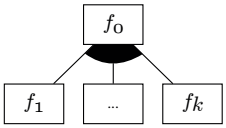
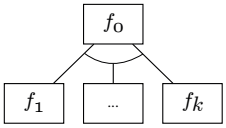
2.4.3 Sur l'équivalence des sémantiques

Il est possible de passer d'une sémantique à une autre, mais ces passages ne sont pas toujours équivalents. Les correspondances entre les trois sémantiques des FMs sont illustrées dans la Figure 2.6.

En haut est représentée la sémantique logique. La petite ellipse représente les formules propositionnelles traduisant la sémantique logique d'un FM. La grande ellipse représente toutes les formules exprimables grâce à la logique des propositions ; cela concerne les formules associées aux FMs, plus toutes les autres formules (φ).

En bas à droite est représentée la sémantique ontologique. La petite ellipse représente tous les FMs pouvant être construits grâce à la notation FODA (correspondant au "diagramme de caractéristiques" dans [SRA+14]), c'est-à-dire l'arbre des caractéristiques, ses décorations et les implications et exclusions mutuelles transverses. La grande ellipse représente tous les FMs accompagnés d'une formule propositionnelle φ quelconque représentant

TABLE 2.2 – Ensemble des relations d'un FM et leur représentation en logique des propositions (restreinte aux connecteurs $\wedge, \vee, \rightarrow, \leftrightarrow, \rightarrow \neg$ et $\oplus$) ; f_i représente le nom d'une caractéristique

Type de relation	**Notation graphique**	**Sémantique logique**
mère-fille		$f_2 \rightarrow f_1$
optionnelle		none
obligatoire		$f_1 \rightarrow f_2$
groupe *or*		$f_0 \rightarrow (f_1 \vee ... \vee f_k)$
groupe *xor*		$f_0 \rightarrow (f_1 \oplus ... \oplus f_k)$
implication binaire	/	$f_1 \rightarrow f_2$
exclusion mutuelle	/	$f_1 \rightarrow \neg f_2$

des contraintes transverses plus complexes (correspondant au "modèle de caractéristiques" dans [SRA+14]). Quatre types de correspondances / transitions sont représentés sous la forme de flèches ; ils sont détaillés ci-dessous.

En bas à gauche est représentée la sémantique de configurations. Cette ellipse représente tous les ensembles de configurations, i.e., tous les ensembles d'ensembles de caractéristiques.

❶ : Nous avons vu précédemment comment traduire la sémantique ontologique d'un FM en une formule propositionnelle représentant sa sémantique logique [Bat05, CW07, PSA+12, SRA+14]. Nous nous plaçons ici dans le cas particulier des formules propositionnelles qui peuvent être obtenues en "traduisant" un FM. Soit $F = \{f_1, \ldots, f_n\}$ un ensemble de variables propositionnelles, une telle formule est définie comme une conjonction d'implications binaires ($f_i \rightarrow f_j$), de mutex ($f_i \rightarrow \neg f_j$), d'équivalences ($f_i \leftrightarrow f_j$) et d'implications ayant pour conclusions un ensemble de disjonctions ($f_1 \vee \ldots \vee f_k$) ou de disjonctions exclusives ($f_1 \oplus \ldots \oplus f_k$). Bien que l'ensemble des connecteurs utilisés par ces formules permettent de définir toutes les fonctions booléennes possibles, les formules propositionnelles construites depuis des FMs n'en représentent qu'un sous-ensemble. Cela est dû au fait qu'il existe des ensembles de configurations qui ne sont pas représentables de manière exacte par un FM [SHTB07].

En suivant la Table 2.2, on peut obtenir plusieurs formules propositionnelles à partir d'un seul FM. Ces formules sont cependant équivalentes, i.e., elles représentent toutes le même ensemble de *modèles* (interprétations les rendant vraies), correspondant aux configu-

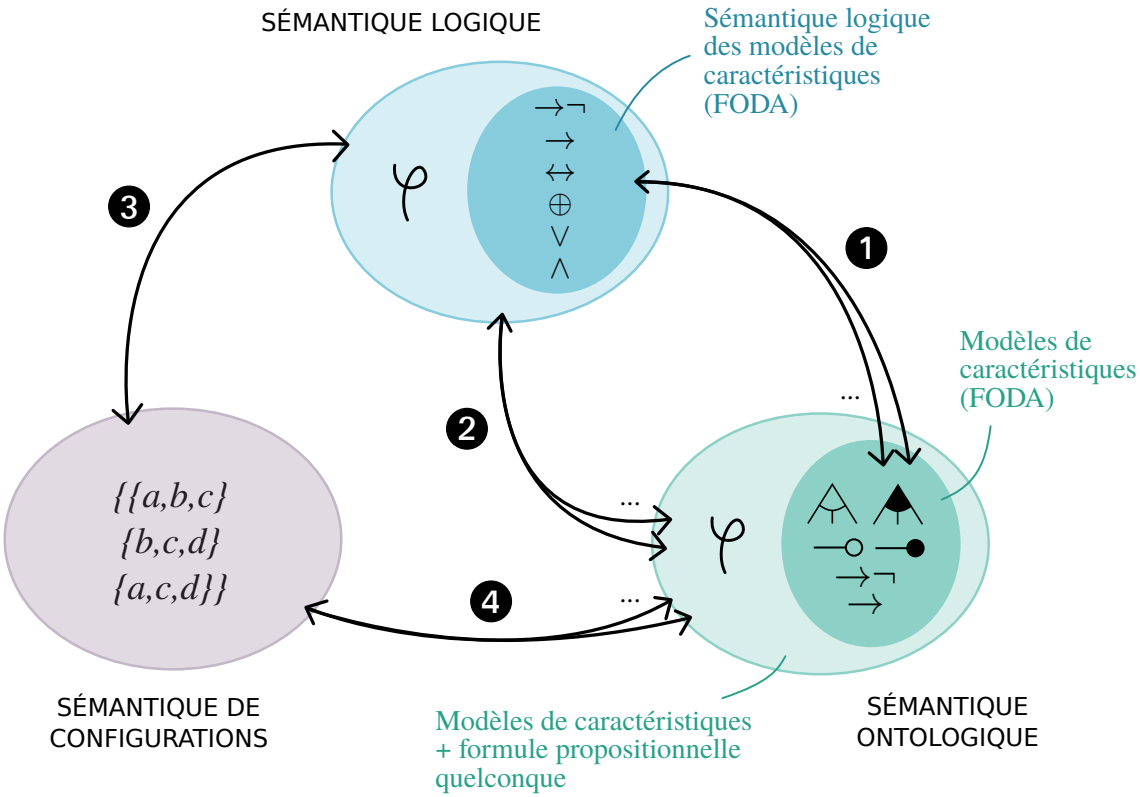


FIGURE 2.6 – Passages entre les trois sémantiques des FMs

rations valides du FM. En écrivant cette formule sous sa forme normale conjonctive (CNF) ou bien sa forme normale disjonctive (DNF), on obtient une représentation canonique de la sémantique logique, i.e., une manière unique d'exprimer la même information. Cette propriété est exprimée dans la Figure 2.6 par la flèche unique côté sémantique logique.

Si plusieurs FMs ont une sémantique ontologique différente (i.e., des arbres de caractéristiques différents), mais qu'ils peuvent être traduits en une formule propositionnelle équivalente (i.e., même CNF et même DNF), ils représentent en réalité la même sémantique logique. On dit que ces FMs sont *différents*, mais *équivalents*. C'est ce qu'indiquent les flèches multiples côté sémantique ontologique dans la Figure 2.6. Si plusieurs FMs différents peuvent avoir la même sémantique logique, cela revient à dire qu'une même formule propositionnelle peut correspondre à plusieurs FMs différents (mais équivalents). Cela est dû au fait que chaque relation des FMs a une seule correspondance en logique propositionnelle, mais que plusieurs de ces relations ont la même traduction. Par exemple, une implication binaire peut correspondre à une relation mère-fille, une relation optionnelle, obligatoire ou encore une implication transverse. Cela rend difficile l'identification d'une sémantique ontologique correcte à partir d'un ensemble de contraintes exprimées par une formule propositionnelle.

En effet, plusieurs travaux ont cherché à passer d'un ensemble de contraintes métiers exprimées sous forme propositionnelle à un FM [CW07, SLB+11, ACP+12, SRA+14] dans le contexte de la ré-ingénierie des LPLs. Seulement, puisqu'il existe potentiellement plusieurs modèles synthétisables depuis une formule propositionnelle, ces méthodes se heurtent à une problématique d'ordre ontologique. Sans l'aide d'un expert du domaine ou d'ontologies externes appropriées [BABN16], il est impossible d'évaluer le sens et la cohérence des arbres de caractéristiques ainsi générés pour en sélectionner le plus pertinent. De plus, dans la majeure partie des cas, les formules propositionnelles de départ sont plus complexes que

celles étudiées ici.

❷ : Nous nous plaçons maintenant dans le cas de toutes les formules exprimables en logique des propositions ($\wedge$, $\vee$, $\neg$). Dans la plupart des cas, au moins une partie est représentable sous la forme d'arbre des caractéristiques, et d'implications et exclusions mutuelles transverses. Le reste est souvent ajouté sous la forme d'une formule propositionnelle quelconque φ en tant que contraintes transverses plus complexes. Les contraintes représentées par cette formule φ sont en général moins compréhensibles par un utilisateur que l'arbre de caractéristiques et les contraintes transverses traditionnelles. C'est pourquoi les travaux cherchant à passer d'une formule à un FM [CW07, SLB+11, ACP+12, SRA+14] tentent de maximiser la partie de la formule de départ représentable par le modèle et donc à minimiser φ en termes de nombre de variables impliquées et de termes de la formule.

Il est possible d'omettre l'ajout de φ aux contraintes transverses, mais dans ce cas-là l'ensemble des configurations valides du modèle synthétisé sera plus vaste que les modèles de la formule propositionnelle de départ, c'est-à-dire que ses interprétations seront toutes incluses dans l'ensemble des configurations valides, mais d'autres configurations qui ne correspondent à aucune interprétation pourront être autorisées par le modèle. Cela pose un problème lorsque le modèle obtenu est utilisé, par exemple, pour faire des opérations automatisées telle que la sélection de produits, car il pourrait permettre à un utilisateur de sélectionner un produit non autorisé par la formule de départ [ABH+13]. C'est pour cette raison que certains travaux préconisent l'ajout d'une formule φ dans les contraintes transverses, ce qui est certainement à l'origine de la distinction entre diagrammes de caractéristiques et FMs dans [SRA+14].

On retrouve dans ce cas-là la même problématique que soulevée dans ❶ : plusieurs modèles équivalents pourront être synthétisés et le plus cohérent du point de vue du domaine restera à identifier.

❸ : Comme vu précédemment, passer de la sémantique logique à la sémantique de configurations nécessite de lister l'ensemble des modèles de la formule propositionnelle. Pour une formule, on n'obtient donc qu'un seul ensemble de configurations possible. Passer d'un ensemble de configurations à une formule propositionnelle les caractérisant produit aussi une représentation canonique, si l'on considère la CNF et la DNF de la formule obtenue.

❹ : Certains travaux se servent de la représentation sous forme logique des FMs pour lister de manière automatique leur ensemble de configurations en utilisant des solveurs SAT [BSRC10, TBK09, MWC09]. Passer d'un ensemble de configurations à une formule propositionnelle est une étape intermédiaire récurrente dans les méthodes proposées pour construire un FM à partir des descriptions de produits logiciels [ACP+12, ABH+13, DDH+13]. Cette formule permet de représenter les interactions entre les caractéristiques de ces descriptions sous la forme de relations logiques, afin de faciliter l'identification des informations sur la variabilité qui sont nécessaires à la construction d'un FM. Cet aspect sera étudié plus en détails dans le chapitre suivant.

La Figure 2.6 illustre bien la problématique liée à la synthèse multiples de FMs différents mais équivalents : que ce soit depuis une représentation de la sémantique logique ou de la sémantique de configurations, une étape de sélection et/ou de filtrage des modèles synthétisés automatiquement est nécessaire pour identifier le plus cohérent. Le passage d'une représentation de la sémantique ontologique vers les autres types de sémantique peut cependant se faire de manière entièrement automatique et produire des représentations canoniques sans qu'aucune décision externe ne soit nécessaire.

Dans ce manuscrit, nous cherchons à synthétiser des FMs depuis des descriptions de produits ; nous travaillons donc à partir d'une sémantique de configurations et nous serons donc aussi confrontés à cette même problématique de synthèse de FMs multiples. Contrairement à beaucoup de méthodes proposées jusqu'ici, nous ne passons pas uniquement par une représentation de la sémantique logique à travers une formule propositionnelle : nous utilisons des structures de treillis conceptuels qui incluent les informations de la sémantique logique, ainsi que la sémantique de configurations et ce de façon canonique. En d'autres termes, nous passons par des structures représentant l'ensemble des configurations et qui mettent en évidence les relations logiques entre les caractéristiques qui définissent ces configurations.

En résumé (2.4) :

- Les **modèles de caractéristiques** sont des modèles de variabilité qui représentent hiérarchiquement un ensemble de **caractéristiques** et des **contraintes** entre ces caractéristiques. Ils décrivent de cette manière un ensemble de **configurations valides** appelé **sémantique de configurations** du modèle.
- Les modèles de caractéristiques sont des représentations qui ne sont **pas canoniques**, c'est-à-dire que plusieurs modèles de caractéristiques **différents** (i.e., ayant des sémantiques ontologiques différentes) peuvent être **équivalents** (i.e., avoir la même sémantique de configurations).
- En plus de la sémantique de configurations, ils possèdent aussi une **sémantique ontologique** et une **sémantique logique**.
- Passer d'une formule propositionnelle représentant une sémantique logique à un ensemble d'ensembles de caractéristiques représentant une sémantique de configurations (et inversement) se fait **sans perdre d'informations**.
- Passer d'une formule propositionnelle ou d'un ensemble d'ensembles de caractéristiques à un arbre de caractéristiques représentant une sémantique ontologique peut aboutir à **plusieurs modèles différents mais équivalents**.
- Si l'on utilise uniquement les éléments de diagramme et les implications et exclusions transverses des FMs de FODA, **toute formule propositionnelle (et donc tout ensemble de configurations) n'est pas représentable**.

La synthèse d'un FM ne se fait pas toujours ex nihilo et doit parfois être réalisée dans le respect d'un ensemble de variantes logicielles existantes. C'est le cas lorsque l'on souhaite opérer une transition depuis une FPL développée sans effort de réutilisation vers une LPL. LPL

2.5 La transition vers des LPLs

Dans cette section, nous discutons du processus de transition vers une LPL à partir d'une FPL. Nous exposons d'abord le contexte et les problématiques qui font émerger plusieurs stratégies d'adoption des LPLs (Section 2.5.1), puis nous détaillons ces stratégies et leur utilisation (Section 2.5.2). Enfin, nous étudions les défis liés à la rétro-ingénierie des LPLs (Section 2.5.3).

2.5.1 Contexte et problématiques

Dans la Section 2.2, nous avons défini une LPL comme la succession de deux processus de développement, i.e., l'ingénierie du domaine et l'ingénierie d'application. Cependant, construire une LPL en suivant ce double processus montre des inconvénients. En effet, ce

n'est qu'après avoir réalisé l'ingénierie du domaine et à la fin de la mise en place de l'ingénierie d'application que les premières variantes logicielles peuvent être dérivées. L'adoption et la mise en place de telles méthodes prend parfois plusieurs années et demande dès le départ un très fort investissement sans profits immédiats [Kru01]. De plus, les effets bénéfiques de la réutilisation ne se font ressentir qu'après le développement de plusieurs variantes logicielles ; Pohl et al. [PBvdL05] fixent ce nombre minimal de variantes à développer à trois. Ils illustrent le coût de développement de n variantes en fonction de leur méthode de développement (i.e., logiciels individuels ou ingénierie des LPLs) par un schéma reproduit en Figure 2.7.

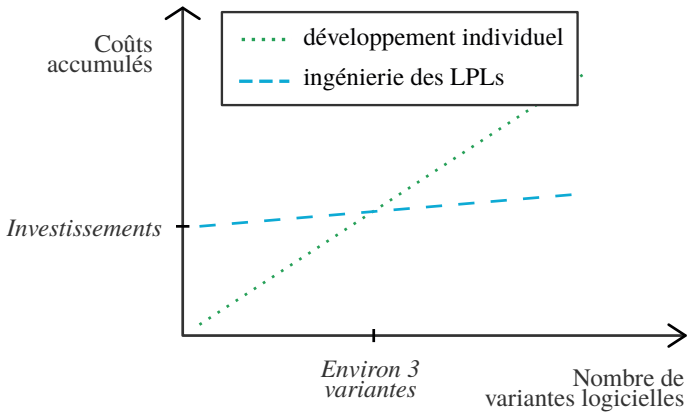


FIGURE 2.7 – Coût de développement de variantes logicielles avec une ligne de produits logiciels et par développement individuel [PBvdL05]

Construire une LPL en suivant ce processus nécessite donc 1) un fort investissement de départ, 2) du temps et de la main d'œuvre et 3) de quoi subsister jusqu'à la dérivation de la 4ème variante logicielle. La construction ex nihilo d'une LPL est donc un investissement à long terme.

De par la difficulté de réunir ces trois conditions, d'autres stratégies d'adoption des LPLs ont fait leur apparition.

2.5.2 Stratégies d'adoption

On peut trouver dans la littérature trois stratégies d'adoption différentes [Kru01, Kru02, ABKS13] :

— La stratégie d'adoption **proactive** suit les deux processus définis en Section 2.2. Dans le cas où les variantes à produire et leurs exigences peuvent être déterminées à l'avance et si celles-ci ne sont pas susceptibles de beaucoup évoluer, alors définir les artefacts et le périmètre de la LPL en amont durant l'analyse et la représentation du domaine peut s'avérer suffisant. Dans ces cas-là, un fort investissement est nécessaire pour la mise en place de l'architecture générique, des artefacts réutilisables et plus généralement de tout le cadre de dérivation au début du projet. Cependant, l'effort de développement est fortement réduit une fois la LPL mise en place.

— Lorsque les exigences de la LPL à produire ne sont pas entièrement déterminées à l'avance, la stratégie d'adoption **réactive** est plus appropriée. C'est une approche incrémentale, qui cherche à développer la LPL au fur et à mesure des nouvelles exigences qui apparaissent sur le marché. Le point de départ peut être un unique système logiciel, à partir duquel sont développées d'autres variantes en augmentant petit à petit le cadre de gestion de la variabilité.

— Enfin, la troisième stratégie d'adoption dite **extractive**, consiste à faire de la rétro-ingénierie d'une LPL depuis un ensemble de variantes existantes. Lorsque plusieurs systèmes logiciels sont développés de manière individuelle et qu'ils présentent des similarités, il peut être utile de les analyser et de s'appuyer sur leurs éléments communs et spécifiques pour extraire des modèles, des architectures et des artefacts sans devoir suivre le processus de construction d'une LPL ex nihilo.

Une enquête [BRN+13] menée sur 35 professionnels ayant travaillé sur des LPLs au sein d'une entreprise montre que l'adoption proactive n'est pas la stratégie la plus utilisée. Dans cette étude, 50% des participants affirment avoir travaillé sur une approche extractive, 47% sur une approche réactive et 35% sur une approche proactive.

Cette tendance s'explique par le fait qu'il est plus facile de développer des variantes individuelles qu'une LPL. Krueger accusait en 2001 le "manque d'outils et de techniques pour construire et pour gérer des collections de variantes logicielles" [Kru01], qui existent cependant en abondance pour le développement de systèmes logiciels individuels. De nombreux outils ont été développés depuis, tels que `pure::variants` [Beu12], `GEARS` [Kru08] ou encore `FeatureIDE` [KTS+09], mais on trouve encore beaucoup de pratiques de réutilisation ad-hoc. On peut donner comme exemple le *clone-and-own* ("cloner et s'approprier") qui est une pratique très répandue dans les entreprises, comme le révèle l'enquête de Dubinsky et al. [DRB+13]. Cela consiste à cloner un système logiciel existant, puis à ajouter ou supprimer des fonctionnalités à ces clones pour créer une nouvelle variante répondant à un nouvel ensemble d'exigences. Certains participants indiquent qu'ils n'ont pas d'autres mécanismes disponibles pour produire des variantes de logiciels existants et témoignent d'un manque d'organisation et de traçabilité en ce qui concerne la réutilisation logicielle.

2.5.3 La rétro-ingénierie des LPLs

Il est donc fréquent qu'un ensemble de variantes logicielles soit développé de manière individuelle et dans ce cas, les professionnels peuvent être confrontés à une double problématique. D'une part, il est difficile de maintenir et de gérer un ensemble de variantes logicielles sans effort de réutilisation, particulièrement lorsque leur nombre, leur taille et leur complexité s'accroissent. Afin de surmonter cette première problématique, nous avons vu que les techniques de réutilisation systématique et de personnalisation de masse sur lesquelles se base l'ingénierie des LPLs forment une approche viable pour gérer des collections de variantes logicielles. D'autre part, mettre en place un processus de transition vers l'ingénierie des LPLs depuis une FPL (i.e., stratégie d'adoption extractive) est une tâche ardue [Kru01].

Notons que la transition vers ce type d'approche n'est pas nécessairement complète. En effet, il y a des cas où la gestion d'une FPL et de sa variabilité peut être grandement améliorée par des méthodes inspirées des LPLs, sans pour autant devoir faire une migration totale vers ce paradigme.

Prenons par exemple le cas de `fork-insight` [RZK18], qui est un outil académique en ligne permettant de regrouper sous forme tabulaire un certain nombre d'informations sur les différents *forks* d'un projet Github. Un des objectifs de cet outil est de faciliter l'analyse et la recherche de variantes d'un projet Github, afin de trouver celles qui correspondent le plus aux attentes de l'utilisateur. Cela fait en partie écho à l'enquête de Dubinsky et al. [DRB+13], qui rapporte qu'il est parfois difficile de trouver la bonne variante à cloner en fonction des exigences requises par la nouvelle. Les initiatives comme `fork-insight` permettent donc de faciliter la gestion de telles variantes, mais pourraient bénéficier des méthodes de sélection et de configuration de produits des LPLs pour aider l'utilisateur dans ses recherches. Un autre exemple est les matrices de comparaison de

produits [BSA+14, SAB13] que l'on trouve en abondance sur internet, notamment sur Wikipedia [2]. Ces matrices représentent des FPL en fonction de leurs caractéristiques sous une forme tabulaire qui facilite leur comparaison. Pouvoir passer de cette représentation extensionnelle vers une représentation intensionnelle complémentaire décrivant la variabilité de la FPL permettrait de donner une vue globale et synthétique à un utilisateur [SAB13].

La migration complète ou partielle d'une FPL vers une LPL est donc un moyen d'améliorer la gestion et la manipulation de la variabilité, dans le but de faciliter la gestion, l'évolution et la maintenance d'un ensemble de variantes logicielles. Nous avons vu précédemment que dans l'ingénierie des LPLs, la plupart des opérations de gestion de la variabilité se basent sur un modèle de variabilité, défini lors de la première étape du processus d'ingénierie du domaine. Construire un tel modèle pour une FPL apparaît naturellement comme la première étape de la migration [Kru01].

Cependant, même avec un nombre restreint de variantes, la construction manuelle d'un tel modèle est un processus long, difficile et faillible. Pour faciliter cette migration, des recherches ont été faites sur la synthèse automatique de FMs [RPK11, ACP+12, LLG+15, AMHS+14, HLE11, BABN16, DDH+13]. En général, ces approches cherchent à extraire des relations correctes et cohérentes entre les caractéristiques qui décrivent les variantes existantes et s'en servent comme base pour construire un modèle représentant la variabilité de la FPL. Cependant, comme nous l'avons vu en Section 2.4, plusieurs modèles différents (i.e., ayant des sémantiques ontologiques différentes) peuvent décrire le même ensemble de variantes. Cette sémantique ontologique des FMs rend donc leur extraction difficile et sujette à produire des modèles incorrects du point de vue du domaine étudié. Baser la gestion de la variabilité de FPLs sur de tels modèles peut provoquer des erreurs importantes [BABN16] rendant plus difficile encore la gestion de ces variantes.

En résumé (2.5) :

- Il existe trois **stratégies d'adoption des LPLs** : **proactive** lorsque la LPL est construite ex nihilo, **réactive** lorsqu'elle est construite incrémentalement selon un système logiciel de départ et **extractive** si la LPL est obtenue par rétro-ingénierie depuis une collection de variantes logicielles existantes.
- De nombreuses FPLs sont développées à partir de **procédés ad-hoc sans organisation de la réutilisation**.
- La migration partielle ou complète vers des LPLs est une solution efficace pour **faciliter la gestion** de ces collections de variantes.
- De ce fait, la stratégie d'adoption extractive est la plus répandue dans les organisations qui adoptent une LPL.
- La transition vers les LPLs est une **tâche ardue** qui nécessite des **techniques et des méthodes pour l'encadrer**.

2.6 Conclusion

Dans ce chapitre, nous avons d'abord défini l'ingénierie des LPLs et les deux processus consécutifs nécessaires à leur construction : l'ingénierie du domaine, durant laquelle sont définis, modélisés et implémentés l'architecture générique et les artefacts réutilisables, puis l'ingénierie d'application, durant laquelle sont mis en place les mécanismes de dérivation des variantes logicielles. Nous avons vu que la documentation et la modélisation de la variabilité sont des tâches cruciales autour desquelles s'articulent l'architecture, les

2. https ://en.wikipedia.org/wiki/Category :Software_comparisons

artefacts et les opérations permettant de créer, de maintenir et de faire évoluer la LPL. Nous avons étudié les deux approches de modélisation de la variabilité les plus connues, i.e., l'approche orientée décisions et l'approche orientée caractéristiques. Alors que la première cherche à modéliser la variabilité du point du vue d'un utilisateur final, la seconde, plus populaire dans la littérature, a pour objectif de modéliser le domaine en termes de caractéristiques, représentant des fonctionnalités de haut niveau, et sert les concepteurs pendant le cycle de vie de la LPL. Nous avons donc étudié plus en détails les FMs, qui sont le standard de facto pour modéliser la variabilité en termes de caractéristiques et plus particulièrement leurs trois sémantiques : ontologique, logique et de configurations. Nous avons établi que les passages entre ces sémantiques n'étaient pas tous réalisables sans perte d'informations : alors que certaines transitions de l'une à l'autre se font de manière automatique, d'autres nécessitent l'intervention d'un expert, notamment pour associer un sens ontologique cohérent à un modèle extrait d'une collection de variantes. Enfin, nous avons étudié les différentes stratégies d'adoption des LPLs. Des enquêtes auprès des professionnels travaillant sur des LPLs en entreprise montrent que l'adoption extractive est la plus répandue. Le long terme imposé par la stratégie proactive semble être un frein à son adoption. Le développement de variantes individuelles dans un premier temps, puis la volonté de transiter vers une approche LPL dans un second temps semblent donc former une stratégie plus répandue, nécessitant de mettre en place un processus de rétro-ingénierie depuis ces variantes existantes. Nous avons vu qu'il pouvait exister une nécessité de migration totale ou bien partielle ; dans les deux cas, des outils et des techniques permettant de gérer la variabilité d'une FPL développée sans effort de réutilisation sont nécessaires pour faciliter leur gestion. L'obtention de FMs à partir de variantes logicielles existantes apparaît comme le nœud du problème, car ils sont le support à la plupart des opérations permettant de simplifier leur exploitation. Les problématiques liées aux équivalences des sémantiques mises en avant précédemment rendent difficile l'obtention de ces modèles et peuvent compromettre les opérations nécessaires à la bonne gestion des variantes.

Pour résumé, la problématique générale est la suivante : **Comment encadrer la migration partielle ou complète d'un ensemble de variantes logicielles existantes, développées sans effort de réutilisation ou de gestion de la variabilité, vers des approches de types LPLs ?**

Dans le reste de ce manuscrit, nous étudions l'analyse formelle de concepts comme un cadre pour faciliter la gestion de la variabilité de FPLs, qui peut supporter des opérations à la manière d'un FM. Alors que ces modèles doivent être rétro-ingénierés, l'analyse formelle de concepts permet d'organiser les variantes existantes dans des structures qui mettent naturellement en évidence la variabilité. Ces structures, bien qu'elles n'explicitent pas d'informations ontologiques comme les FMs, représentent tout de même la variabilité sous forme logique et structurelle. Les opérations sur la variabilité que l'AFC permet de réaliser incluent entre autres la recherche d'information, la représentation hiérarchique, ou encore l'explicitation d'architecture. Bien qu'elles n'aient pas toutes les capacités des FMs en termes de visualisation et de sémantique ontologique, les structures de l'analyse formelle de concepts permettent tout de même de manipuler, d'analyser et de gérer la variabilité de variantes existantes jusqu'à un certain point et ne souffrent pas des problèmes de rétro-ingénierie lors de la migration. Nous pensons que ce cadre est prometteur pour faciliter la migration partielle ou complète vers une LPL. Le chapitre suivant traite des différentes informations sur la variabilité qui sont naturellement mises en évidence par l'analyse formelle de concepts.

Chapitre 3

L'analyse formelle de concepts : un cadre mathématique structurel pour la représentation de la variabilité

Préambule

Dans le Chapitre 2, nous avons étudié les problématiques liées à la rétro-ingénierie des lignes de produits logiciels. Nous avons notamment analysé les problématiques liées à la construction d'un modèle de variabilité représentatif et correct basé sur des variantes logicielles existantes, car de tels modèles servent de support à la majorité des traitements associés aux lignes de produits logiciels. Dans ce chapitre, nous étudions l'analyse formelle de concepts, un cadre mathématique structurel pour l'analyse de données, la gestion d'informations et la représentation des connaissances, ainsi que ses structures conceptuelles associées pour représenter la variabilité d'une famille de produits. Dans un premier temps, nous définissons l'analyse formelle de concepts ainsi que quatre de ses structures conceptuelles, qui organisent naturellement des éléments en fonction des caractéristiques qu'ils partagent. Nous identifions ensuite les informations sur la variabilité qui sont naturellement mises en évidence dans ces structures et nous proposons une méthode correcte et complète permettant de les extraire. Enfin, nous lions les informations extraites aux modèles de caractéristiques et montrons ainsi que les structures de base de l'analyse formelle de concepts représentent des classes d'équivalence de modèles de caractéristiques.

Sommaire

3.1 Introduction

L'analyse formelle de concepts (AFC) [GW99] est un cadre mathématique pour l'analyse de données, la représentation des connaissances et la gestion d'informations [Pri06]. À partir d'un ensemble fini d'objets décrits par des attributs binaires, l'AFC construit des structures conceptuelles canoniques organisant les objets en fonction de leurs attributs communs. L'AFC a été appliquée à de nombreux domaines, incluant notamment les ontologies [BNT08, MLNC17, OSS04, NSKS06, BGSS07], l'extraction de connaissances [PEVD10, LS05], ou encore l'ingénierie logicielle [RPK11, LP07, Huc15, GM93, Sne00, TCBE05, AMSH$^+$13]. Les structures conceptuelles ont été entre autres utilisées dans l'ingénierie des lignes de produits logicielles [NE09, SSD13], notamment dans la gestion de la variabilité [RPK11, LP07, AMHS$^+$14, SSS17, AMdSH$^+$14] ; les objets représentent alors les configurations des variantes logicielles et les attributs représentent les caractéristiques. En organisant les objets en fonction de leurs attributs communs, les structures conceptuelles mettent naturellement en évidence ce qui est commun et ce qui est variable dans l'ensemble de données de départ [AMdSH$^+$14], ce qui fait d'elles des structures pertinentes pour représenter la variabilité en termes de caractéristiques [CBHN16].

Dans ce chapitre, nous étudions les qualités de l'AFC et de ses structures conceptuelles pour la représentation de la variabilité. L'AFC est un cadre structurel général : ses structures ne sont pas construites dans un but fonctionnel ; leur construction répond à des propriétés mathématiques mettant naturellement en évidence des informations. Dans ce qui suit, nous analysons ces informations et déterminons ce qu'elles représentent du point de vue de la variabilité. De plus, ces structures contenant à la fois les configurations et les caractéristiques, c'est une représentation à la fois intensionnelle et extensionnelle de la variabilité. **Nous défendons l'idée que l'AFC est un cadre idéal pour représenter la variabilité d'un ensemble de variantes existantes.**

Notre étude se déroule comme suit. Nous commençons par définir théoriquement l'AFC ainsi que quatre de ses structures conceptuelles (Section 3.2). Puis nous étudions plusieurs aspects de ces structures, tels que les concepts et leur place dans la structure et nous documentons les informations que ces aspects donnent sur la variabilité de l'ensemble de variantes considéré (Section 3.3). Nous montrons ensuite comme extraire ces informations sous forme de relations logiques, qui peuvent être réutilisées dans des applications futures de gestion de variabilité. Nous menons une étude comparative des différentes méthodes d'extraction de variabilité existantes et nous montrons que l'AFC permet de faire au moins aussi bien que toutes ces méthodes et ce en s'appuyant sur une seule structure (Section 3.4). Enfin, nous établissons une comparaison entre les structures conceptuelles et les FMs étudiés au chapitre précédent. Nous montrons comment une structure conceptuelle donnée inclut tous les FMs ayant la même sémantique de configurations et en donne ainsi une représentation canonique (Section 3.5).

3.2 L'analyse formelle de concepts : définitions

L'analyse formelle de concepts (AFC) [GW99] est un cadre mathématique qui réalise une classification d'un ensemble d'objets décrits par des attributs. La Figure 3.1 synthétise le fonctionnement de l'AFC.

1. Contexte formel

En données d'entrée, l'AFC se base sur un *contexte formel* représentant l'ensemble des objets en fonction des attributs qu'ils possèdent. Les attributs manipulés ici sont binaires (étant donné un attribut, un objet peut le posséder ou non).

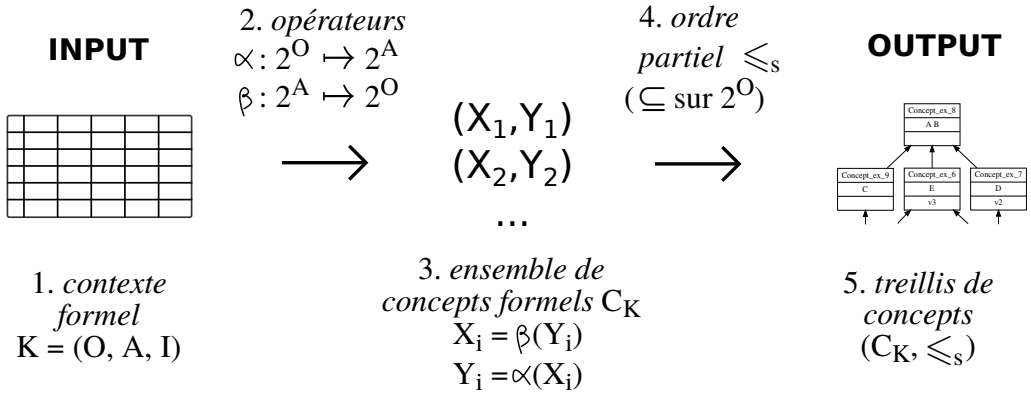


FIGURE 3.1 – Synthèse du fonctionnement de l'analyse formelle de concepts

Définition 3.1 (Contexte formel)**.** *Un contexte formel est un triplet $K = (O, A, I)$ tel que O est un ensemble fini d'objets, A est un ensemble fini d'attributs et $I \subseteq O \times A$ est une relation d'incidence définissant quels objets possèdent quels attributs.*

Un contexte formel peut être exprimé sous la forme d'une table binaire, dans laquelle les lignes sont les objets dans O et les colonnes sont les attributs dans A. Chaque couple $(o, a) \in I$ est indiqué par une croix dans la cellule correspondant à l'objet o et à l'attribut a. La Table 3.1 présente un exemple de contexte formel, élaboré depuis une matrice de comparaison de produits de wikipedia [1]. Il présente 5 objets qui sont des outils de modélisation de données. Ces objets sont décrits par 9 attributs, déterminant s'ils possèdent au moins un système d'exploitation (`Operating System`), s'ils peuvent gérer des modèles de données (`Data Model`) et représentant les systèmes d'exploitation (`OS:`) avec lesquels ils sont compatibles, ainsi que les modèles de données (`DM:`) qu'ils peuvent gérer.

TABLE 3.1 – Un contexte formel K_{DM} décrivant 5 objets (ici des outils de modélisation de données) en fonction de 9 attributs (ici représentant des systèmes d'exploitation et des modèles de données)

DM_ tools (K_{DM})	`Operating System`	`Data Model`	`OS:Windows`	`OS:Mac OS`	`OS:Linux`	`DM:Conceptual`	`DM:Physical`	`DM:Logical`	`DM:ETL`
Astah	×	×	×	×	×	×			
Erwin DM	×	×	×			×	×	×	
ER/Studio	×	×	×			×	×	×	×
Magic Draw	×	×	×	×	×	×	×	×	
MySQL Workbench	×	×	×	×	×		×		

Par exemple, on peut lire dans cette table que :
— l'objet *MySQL Workbench* possède les quatre attributs `Operating System`, `Data Model`, `OS:Windows`, `OS:Mac OS`, `OS:Linux` et `DM:Physical` ;
— l'attribut `DM:Logical` est partagé par les trois objets *Erwin DM*, *ER/Studio* et *Magic Draw*.

1. https ://en.wikipedia.org/wiki/Comparison_of_data_modeling_tools, dernier accès en février 2018

2. Opérateurs de dérivation

L'AFC définit deux opérateurs de dérivation $\alpha : 2^O \mapsto 2^A$ et $\beta : 2^A \mapsto 2^O$ tels que :

$$\alpha(X) = \{a \in A | \forall o \in X, (o, a) \in I\}, \quad \text{avec } X \subseteq O$$
$$\beta(Y) = \{o \in O | \forall a \in Y, (o, a) \in I\}, \quad \text{avec } Y \subseteq A$$

L'opérateur α associe à un ensemble d'objets de O l'ensemble de tous les attributs qu'ils partagent. L'opérateur β associe à un ensemble d'attributs de A l'ensemble de tous les objets qui partagent ces attributs.

3. Concepts formels

L'application de l'AFC sur un contexte formel K extrait un ensemble fini C_K de *concepts formels*, en se basant sur les deux opérateurs définis précédemment.

Définition 3.2 (Concept formel)**.** *Un concept formel est une paire $C = (X, Y)$, avec $X \subseteq O$ et $Y \subseteq A$, vérifiant $\alpha(X) = Y$ et $\beta(Y) = X$. On appelle X l'extension du concept et Y l'intension du concept.*

Un concept formel représente un ensemble maximal d'objets partageant un ensemble maximal d'attributs, c'est-à-dire qu'il n'y a pas d'autres objets que ceux de X qui partagent tous les attributs de Y et qu'il n'y a pas d'autres attributs que ceux de Y qui soient partagés par tous les objets de X. Graphiquement, un concept correspond à un rectangle de croix de taille maximale dans le contexte formel, aux permutations près des lignes et des colonnes.

Exemple. Considérons le contexte K_{DM} de la Table 3.1. Prenons l'ensemble d'objets {*Erwin DM*, *ER/Studio*} : ils partagent tous deux l'ensemble d'attributs {`Operating System`, `Data Model`, `OS:Windows`, `DM:Conceptual`, `DM:Physical`, `DM:Logical`}. À présent, complétons notre ensemble d'objets pour prendre en compte tous les objets partageant cet ensemble d'attributs : on obtient cette fois-ci les objets {*Erwin DM*, *ER/Studio*, *Magic Draw*}. La paire {{*Erwin DM*, *ER/Studio*, *Magic Draw*}, {`Operating System`, `Data Model`, `OS:Windows`, `DM:Conceptual`, `DM:Physical`, `DM:Logical`}} constitue alors un concept formel du contexte K_{DM}. {*Erwin DM*, *ER/Studio*, *Magic Draw*} représente l'extension de ce concept et {`Operating System`, `Data Model`, `OS:Windows`, `DM:Conceptual`, `DM:Physical`, `DM:Logical`} représente l'intension de ce concept.

Une opération similaire peut être effectuée en partant d'un ensemble d'attributs.

4. Ordre partiel

L'AFC définit un ordre partiel $\leq_s$ sur l'ensemble des concepts formels C_K de K. Cet ordre est basé sur l'inclusion ensembliste des extensions des concepts et de manière duale, sur l'inclusion inverse de leurs intensions.

Définition 3.3 (Ordre partiel $\leq_s$)**.** *Soient deux concepts $C_1 = (X_1, Y_1)$ et $C_2 = (X_2, Y_2)$ de C_K, $C_1 \leq_s C_2$ si est seulement si $X_1 \subseteq X_2$ (ce qui équivaut par construction à $Y_2 \subseteq Y_1$). C_1 est appelé sous-concept de C_2 et C_2 super-concept de C_1.*

Dans de nombreux travaux sur l'héritage [GM93, Huc15], cet ordre partiel est aussi appelé *ordre de spécialisation*. En effet, on dit que les concepts sont ordonnés par spécialisation car un sous-concept possède tous les attributs de ses super-concepts, avec éventuellement d'autres attributs supplémentaires.

Exemple. Le concept {{*ER/Studio*}, {`Operating System`, `Data Model`, `OS:Windows`, `DM:Conceptual`, `DM:ETL`, `DM:Physical`, `DM:Logical`}} est un sous-concept du concept extrait dans l'exemple précédent : il possède un sous-ensemble des objets du précédent concept, ainsi que tous ses attributs, plus `DM:ETL`.

5. Treillis de concepts, et autres structures conceptuelles

L'ensemble des concepts formels C_K partiellement ordonnés par l'ordre de spécialisation $\leq_s$ forme une structure de *treillis*, appellée *treillis de concepts*. Pour définir la notion de treillis, puis de treillis de concepts, nous définissons d'abord les notions de *borne inférieure* et de *borne supérieure*.

Définition 3.4 (Borne inférieure)**.** *La borne inférieure d'un ensemble d'éléments F appartenant à un ordre partiel E est un unique élément parmi les minorants de F (i.e. tous les éléments de E qui sont inférieurs à tous les éléments de F) qui est plus grand que tous les autres.*

Définition 3.5 (Borne supérieure)**.** *La borne supérieure d'un ensemble d'éléments F appartenant à un ordre partiel E est un unique élément parmi les majorants de F (i.e. tous les éléments de E qui sont supérieurs à tous les éléments de F) qui est plus petit que tous les autres.*

Définition 3.6 (Treillis)**.** *Un treillis est un ensemble partiellement ordonné dans lequel chaque paire d'éléments possède une borne inférieure et une borne supérieure.*

Définition 3.7 (Treillis de concepts)**.** *Soit un contexte formel K et son ensemble de concepts formels C_K. L'ensemble C_K muni de l'ordre de spécialisation $\leq_s$ forme une structure de treillis appelée treillis de concepts.*

La Figure 3.2 présente le diagramme de Hasse du treillis de concepts construit à partir du contexte formel de la Table 3.1, duquel 10 concepts ont été extraits. Le diagramme de Hasse est une représentation graphique d'un ensemble ordonné et fini respectant certaines règles visant à en faciliter la visualisation :

— chaque élément est représenté par un point ou un élément graphique ;
— une ligne entre deux éléments représente leur relation d'ordre et seule la réduction transitive est dessinée (par rapport à un diagramme de Hasse traditionnel, ici nous mettons des flèches et non des lignes) ;
— si un élément est plus grand qu'un autre, il apparaît plus haut dans le diagramme.

Avec la représentation choisie dans cette thèse, un concept est représenté sous la forme d'un rectangle partagé en trois parties : la partie supérieure donne le nom du concept, qui est unique dans le treillis, la partie du milieu représente l'intension du concept et la partie inférieure son extension. Une flèche partant d'un concept vers un autre représente l'ordre partiel entre ces concepts : une flèche partant d'un concept C_1 vers un concept C_2 signifie que $C_1 \leq_s C_2$.

Dans ce treillis, l'extension et l'intension des concepts sont représentées de manière simplifiée ; en effet, chaque attribut (resp. chaque objet) n'apparaît qu'une fois dans la structure. Un attribut apparaît uniquement dans le plus haut concept de la structure possédant cet attribut. À l'inverse, un objet apparaît uniquement dans le concept le plus bas de la structure possédant cet objet. Puisqu'un concept hérite de tous les attributs de ses super-concepts et de tous les objets de ses sous-concepts, l'extension et l'intension complètes de chaque concept peuvent être reconstituées par héritage. Un concept ayant au moins un attribut dans son intension simplifiée est appelé *introducteur d'attribut*. Un concept ayant au moins un objet dans son extension simplifiée un *introducteur d'objet*. Les concepts qui ne sont ni des introducteurs d'attributs, ni des introducteurs d'objets sont appelés des *concepts neutres*.

Définition 3.8 (Concept introducteur)**.** *Soit un contexte formel $K = (O, A, I)$.*

Un concept $C = (X, Y), C \in C_K$, introduit l'attribut $a \in A$ si et seulement si ($a \in Y$) et ($\nexists C' = (X', Y'), C' \in C_K | a \in Y'$ et $C \leq_s C'$). On a $X = \beta(\{a\})$ et $Y = \alpha \circ \beta(\{a\}) = \alpha(\beta(\{a\}))$. On appelle ces concepts des ***concepts introducteurs d'attributs****.*

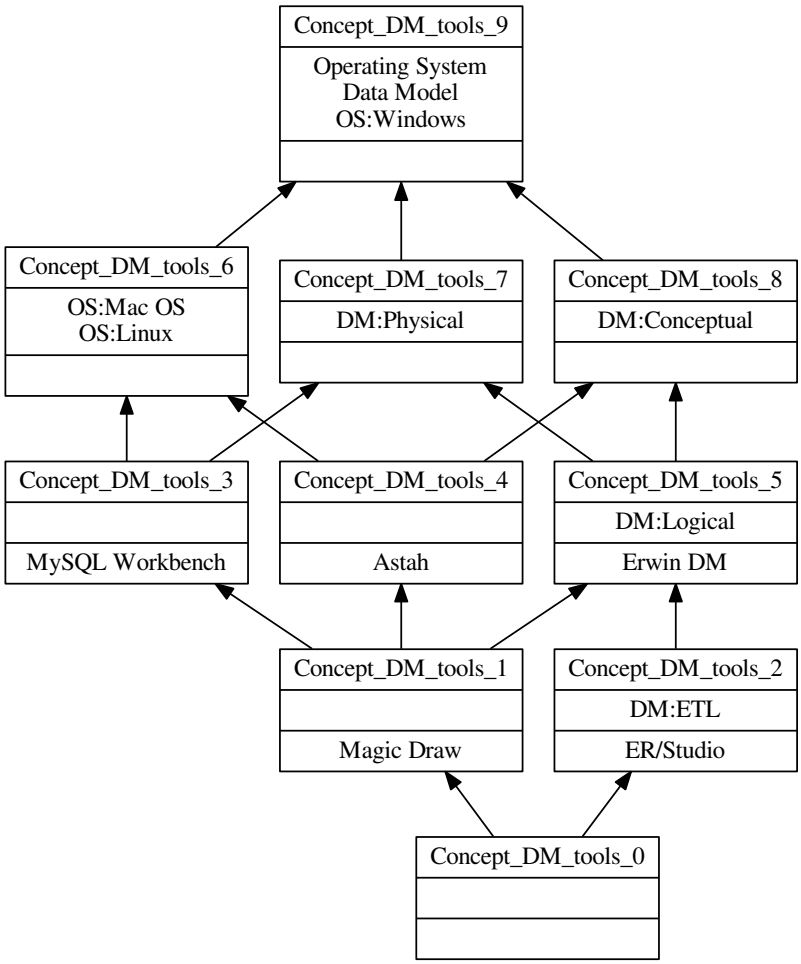


FIGURE 3.2 – Treillis de concepts de la Table 3.1

Un concept $C = (X, Y), C \in C_K$*, introduit l'objet* $o \in O$ *si et seulement si* $(o \in X)$ *et* $(\nexists C' = (X', Y'), C' \in C_K | o \in X'$ *et* $C' \leq_s C)$*. On a* $X = \beta \circ \alpha(\{o\}) = \beta(\alpha(o))$ *et* $Y = \alpha(\{o\})$*. On appelle ces concepts des* ***concepts introducteurs d'objets****.*

Exemple. Dans la Figure 3.2, le `Concept_DM_tools_4` introduit l'objet *Astah* et le `Concept_DM_tools_7` introduit l'attribut `DM:Physical`. Le `Concept_DM_tools_5` introduit l'attribut `DM:Logical` et l'objet *Erwin DM* et le `Concept_DM_tools_0` n'introduit aucun élément, il est donc neutre.

Par la suite, on nommera OC_K et AC_K respectivement l'ensemble des concepts introducteurs d'objets et l'ensemble des concepts introducteurs d'attributs d'un contexte K. Ces deux ensembles ne sont pas nécessairement disjoints.

Dans certaines applications, il n'est pas nécessaire de prendre en compte les concepts neutres, par exemple si l'objet de la recherche est la hiérarchie entre les éléments et non pas les groupes de ces éléments. Dans le premier cas, il est possible d'utiliser un sous-ordre du treillis des concepts, restreint à l'ensemble des concepts introducteurs. Cette sous-structure est appelée "l'ensemble partiellement ordonné des concepts introducteurs d'attributs et des concepts introducteurs d'objets" (en anglais *Attribute-Object-Concept partially ordered set*, abrégé en **AOC-poset**) [Pet01]. Contrairement au treillis de concept, l'AOC-poset n'est pas nécessairement un treillis.

Définition 3.9 (AOC-poset)**.** *Soit* C_K *l'ensemble de tous les concepts extraits d'un contexte*

formel K, on appelle AOC-poset le sous-ordre de $(C_K, \leq_s)$ restreint aux concepts introducteurs : $(C'_K, \leq_s)$ tel que $C'_K = OC_K \cup AC_K$,

La Figure 3.3 représente l'AOC-poset associé au contexte formel de la Table 3.1. Dans cet exemple, il ne perd que le concept en bas du treillis (ici `Concept_DM_tools_0`, qui est le seul concept neutre du treillis précédent), mais dans le cas général, les concepts neutres peuvent apparaître n'importe où dans le treillis.

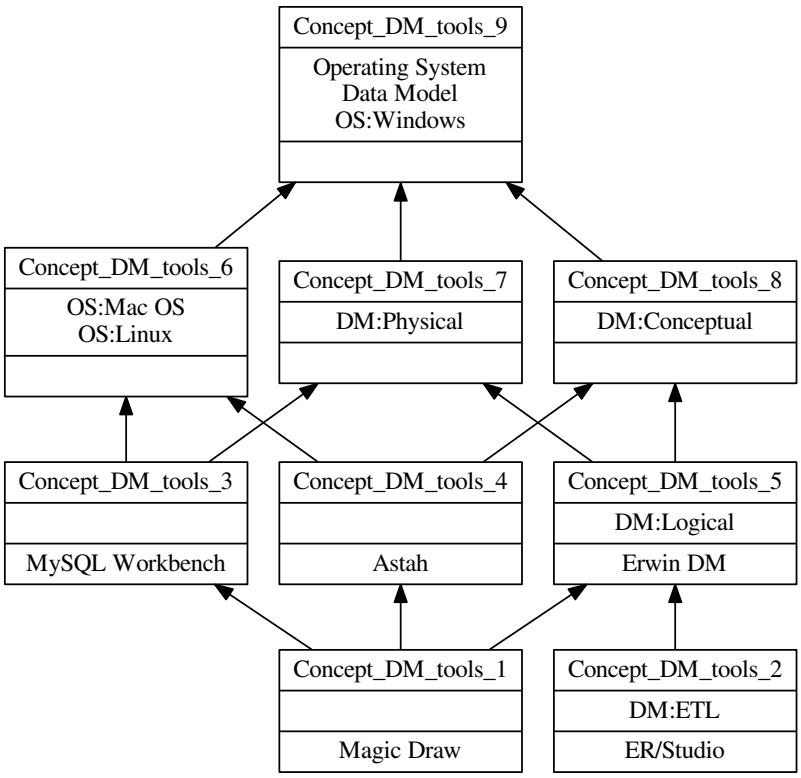


FIGURE 3.3 – AOC-poset associé avec le contexte formel de la Table 3.1

De la même façon, on peut garder uniquement le sous-ordre restreint aux concepts introducteurs d'attributs (*attribute-concept partially ordered set*, abrégé en **AC-poset**), ou aux concepts introducteurs d'objets (*object-concept partially ordered set*, abrégé en **OC-poset**).

Définition 3.10 (AC-poset)**.** *Soit C_K l'ensemble de tous les concepts extraits d'un contexte formel K, on appelle **AC-poset** le sous-ordre de $(C_K, \leq_s)$ restreint aux concepts introducteurs d'attributs : $(AC_K, \leq_s)$.*

Définition 3.11 (OC-poset)**.** *Soit C_K l'ensemble de tous les concepts extraits d'un contexte formel K, on appelle **OC-poset** le sous-ordre de $(C_K, \leq_s)$ restreint aux concepts introducteurs d'objets : $(OC_K, \leq_s)$.*

La Figure 3.4 représente l'AC-poset et l'OC-poset associés au contexte formel de la Table 3.1.

Notons que dans l'AC-poset, on perd la factorisation des objets, i.e. un objet peut être introduit dans plusieurs concepts. Ceci est dû au fait que les introducteurs d'objets n'apparaissent pas toujours dans cette structure. Par exemple, dans l'AC-poset de la Figure 3.4, l'objet *Astah* est introduit dans les concepts 6 et 8. Respectivement, dans l'OC-poset, on perd la factorisation des attributs. Par exemple, l'atribut `DM_conceptual` est introduit deux fois dans l'OC-poset de la Figure 3.4, dans les concepts 4 et 5.

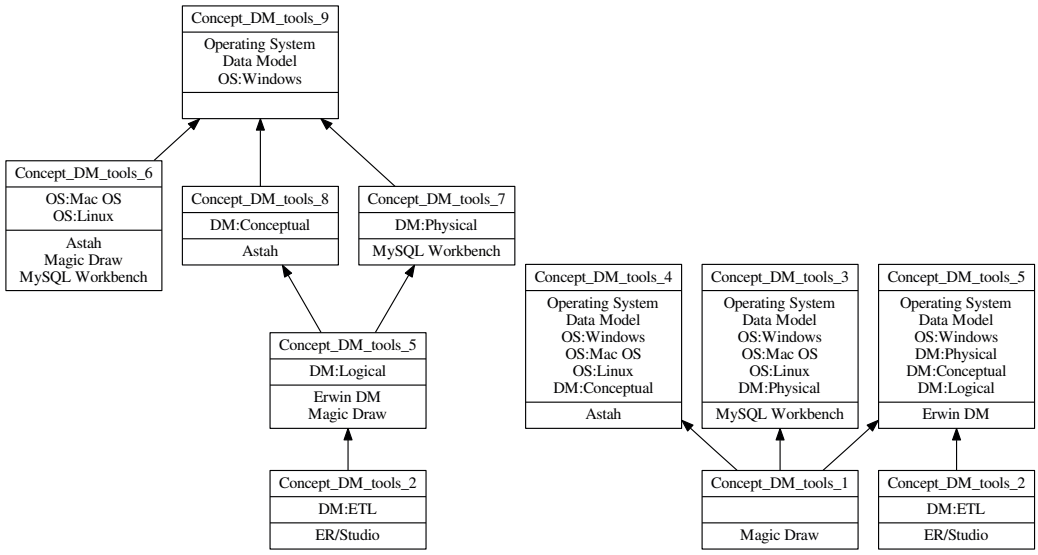


FIGURE 3.4 – AC-poset (gauche) et OC-poset (droite) associés au contexte formel de la Table 3.1

Un dernier point important concerne la taille des structures conceptuelles. Les treillis de concepts ont un nombre de concepts qui grandit exponentiellement avec la taille des données d'entrée : la borne supérieure de ce nombre est de $2^{min(|O|,|A|)}$. Cependant, l'AOC-poset, l'OC-poset et l'AC-poset perdent cette croissance exponentielle et ont une taille bornée supérieurement par $|O| + |A|$, $|O|$ et $|A|$, respectivement.

En résumé (3.2) :

- L'**analyse formelle de concepts** est un cadre mathématique de groupement hiérarchique pour l'analyse de données, la représentation de connaissance et la gestion d'informations.
- Les données d'entrée s'expriment sous la forme d'un **contexte formel**, une table représentant un ensemble d'objets en fonction des attributs qu'ils possèdent.
- Du contexte formel sont extraits des **concepts formels**, représentant des ensembles maximaux d'objets partageant des ensembles maximaux d'attributs.
- Organisés par spécialisation, ces concepts forment une structure de treillis unique appelée **treillis de concepts**.
- Il existe plusieurs types de concepts, selon qu'ils introduisent des objets, des attributs, les deux, ou bien aucun des deux.
- Des sous-hiérarchies restreintes à certains types de concepts, telles que l'**AOC-poset**, l'**AC-poset** et l'**OC-poset**, peuvent être utilisées à la place du treillis dans certaines applications, permettant ainsi de limiter la taille de la structure manipulée.

Dans ce qui suit, nous étudions certains aspects des quatre structures conceptuelles que nous venons de définir (i.e., treillis de concepts, AOC-poset, AC-poset et OC-poset), comme les concepts formels et la relation de spécialisation, afin d'identifier les informations sur la variabilité qu'ils mettent en avant.

3.3 Les structures conceptuelles : un espace de représentation de la variabilité

Dans les structures conceptuelles, les concepts formels ainsi que l'ordre de spécialisation entre ces concepts mettent en évidence des propriétés qui sont vraies sur l'ensemble des objets étudiés. Dans cette section, nous étudions plus spécifiquement les propriétés concernant la variabilité que l'on peut y lire. Pour faciliter le lien avec le vocabulaire de la variabilité, nous utiliserons le terme "caractéristiques" pour désigner les attributs et "configurations" pour désigner les objets. Nous étudions les treillis de concepts, ainsi que trois de leurs sous-ordres : les AOC-posets, les AC-posets et les OC-posets.

Dans ce qui suit, nous étudions les informations contenues dans les concepts formels (Section 3.3.1), celles qui sont issues de la place de ces concepts les uns par rapport aux autres (Section 3.3.2) et de la place des concepts dans la structure (Section 3.3.3).

3.3.1 Les concepts formels

L'AFC est une formalisation mathématique d'une théorie commune aux domaines de la psychologie, de la philosophie et de la logique, appelée *théorie classique des concepts*. Cette théorie stipule qu'un concept est défini par un ensemble d'attributs qu'un objet se doit de posséder pour appartenir à ce concept. Les concepts formels de l'AFC sont des entités mathématiques qui découlent de cette théorie. Les termes *intension* (attributs / caractéristiques du concept formel) et *extension* (objets / configurations du concept formel) utilisés dans cette formalisation proviennent aussi des domaines de la philosophie et de la logique. Dans ce qui suit, nous étudions d'abord les concepts introducteurs, puis les concepts neutres.

Les concepts introducteurs

Par construction, un concept formel représente un ensemble maximal de configurations partageant un ensemble maximal de caractéristiques. Soit $K = (O, A, I)$ un contexte formel, un concept formel est donc une paire $(X, Y), X \subseteq O, Y \subseteq A$ telle qu'il n'y a pas d'autres configurations que celles de X qui partagent toutes les caractéristiques de Y et qu'il n'y a pas d'autres caractéristiques que celles de Y qui soient partagées par les configurations de X. Le treillis de concepts regroupe tous les concepts pouvant être extraits d'un contexte formel.

Les trois affirmations suivantes sont donc équivalentes :
- il existe un concept dans le treillis tel que ce concept possède une caractéristique a dans son intension et qu'il n'existe pas de plus grand ensemble de configurations que celui correspondant à l'extension de ce concept qui possède a ;
- ce concept est le concept qui possède a dans son intension et qui est le plus haut dans le diagramme de Hasse du treillis de concepts ;
- ce concept introduit la caractéristique a.

En effet, il n'existe qu'un seul concept $C = (X, Y)$ tel que l'ensemble $X = \beta(\{a\})$ représente toutes les configurations possédant la caractéristique a. Il ne peut donc pas exister de plus grand ensemble de configurations ayant a que celui correspondant à X. De plus, les concepts étant organisés par l'inclusion ensembliste sur leurs extensions dans le diagramme de Hasse, les concepts ayant les extensions les plus grandes sont placés plus haut dans le treillis. Le concept $C = (\beta(\{a\}), \alpha(\beta(\{a\})))$ est donc le plus haut concept du treillis ayant a dans son intension.

Du point de vue de la variabilité, on peut donc interpréter un concept introducteur de caractéristiques ainsi :

Propriété 3.1. *L'extension d'un concept introducteur de caractéristiques concentre toutes les configurations possédant les caractéristiques qu'il introduit.*

De manière duale, on peut affirmer les trois propositions suivantes :
— il existe un concept dans le treillis tel que ce concept possède une configuration o dans son extension et qu'il n'existe pas de plus grand ensemble de caractéristiques décrivant o que celui correspondant à l'intension de ce concept ;
— ce concept est le concept qui possède o dans son extension le plus bas dans le diagramme de Hasse du treillis de concepts ;
— ce concept introduit la configuration o.

En effet, il n'existe qu'un seul concept $C = (X, Y)$ tel que l'ensemble $Y = \alpha(\{o\})$ représente toutes les caractéristiques décrivant la configuration o. Il ne peut donc pas exister de plus grand ensemble de caractéristiques décrivant o que celui de Y. Les concepts étant organisés par l'inclusion ensembliste inverse sur leurs intensions dans le diagramme de Hasse, les concepts ayant les intensions les plus grandes sont placés plus bas dans le treillis. Le concept $C = (\beta(\alpha(\{o\})), \alpha(\{o\}))$ est donc le plus haut concept du treillis ayant o dans son extension.

Un concept introducteur de configuration peut donc être interprété de la façon suivante :

Propriété 3.2. *L'intension d'un concept introducteur de configuration représente l'ensemble exact des caractéristiques composant cette configuration.*

Ces deux propriétés sont vraies dans les trois sous-ordres du treillis des concepts, lorsque celles-ci s'appliquent (e.g., la propriété sur les introducteurs de configurations ne s'applique pas toujours à l'AC-poset car il ne possède que les introducteurs de caractéristiques).

Les concepts neutres

Dans ce qui suit, nous nous intéressons aux concepts neutres (i.e. non-introducteurs), qui ne sont présents que dans le treillis des concepts.

Nous définissons une **configuration invalide** comme étant une combinaison de caractéristiques interdite, telle qu'il n'existe pas de séquence d'ajout de caractéristiques rendant la configuration valide.

Exemple. La combinaison de caractéristiques {`Operating System`, `OS:Linux`, `DM:ETL`} est invalide car dans l'ensemble des configurations étudié, seule *ER/Studio* possède `DM:ETL`, mais cette configuration ne possède pas `OS:Linux`. Aucun ajout de caractéristiques à cette combinaison ne peut donc aboutir à une configuration valide : la combinaison initiale est donc invalide.

Une **configuration partielle** est une configuration qui n'est pas valide, mais qui peut le devenir si l'on y rajoute certaines caractéristiques.

Exemple. La combinaison de caractéristiques {`Operating System`, `Data Model`, `OS:Windows`, `OS:Mac OS`} ne correspond à aucune configuration valide de la Table 3.1. Cependant, si l'on rajoute les caractéristiques `OS:Linux` et `DM:Conceptual`, on obtient la configuration valide associée à la variante *Astah* : la combinaison initiale est donc partielle. Notons aussi que l'ajout de `OS:Linux` et de `DM:Physical` à la combinaison initiale correspond cette fois à la configuration valide associée à la variante *MySQL Workbench* ; la configuration partielle est ici partagée par plusieurs configurations valides.

Nous savons que les intensions des concepts représentent des ensembles de caractéristiques qui sont communs à un ensemble de configurations ; par construction, ces ensembles de caractéristiques ne représentent alors pas de combinaisons interdites et donc de configurations invalides.

Propriété 3.3. *L'intension d'un concept introducteur de configuration représente une configuration valide et l'intension d'un concept n'introduisant pas de configuration représente une configuration partielle.*

En plus de respecter toutes les contraintes d'exclusion mutuelle vraies dans le contexte formel, les concepts respectent également toutes les implications binaires et donc aussi toutes les cooccurrences (i.e. doubles implications binaires). Ces types de contraintes peuvent être appliquées automatiquement sur une combinaison de caractéristiques, i.e. pour la compléter grâce à un mécanisme d'inférence afin qu'elle respecte l'ensemble des contraintes de départ. Parmi les 5 types de contraintes exprimés dans les FMs, les exclusions mutuelles, les implications binaires et les cooccurrences représentent l'ensemble des contraintes pouvant être appliquées automatiquement sur une configuration. Elles sont différentes des deux autres types de contraintes exprimant des groupes de caractéristiques, car ces derniers nécessitent un choix de l'utilisateur. On dit qu'une configuration partielle est maximale s'il n'est pas possible de la compléter davantage de manière automatique, mais qu'elle n'est pas encore valide.

Définition 3.12 (Configuration partielle maximale)**.** *Soit F un ensemble de caractéristiques. Soit IS, C et M trois ensembles respectivement d'implications binaires, d'équivalences et d'exclusions mutuelles sur F définits comme suit :*

- *$imp \in IS$ est une implication binaire $(a \Rightarrow b)$ avec $a, b \in F$*
- *$cooc \in C$ est une équivalence $(a \Leftrightarrow b)$ avec $a, b \in F$*
- *$mut \in M$ est une exclusion mutuelle $(a \Rightarrow \neg b)$ avec $a, b \in F$*

Soit $c \subseteq F$ un sous-ensemble de caractéristiques de F et i_c une interprétation sur les variables logiques correspondant aux caractéristiques de F telle que $\forall c_j \in c,\ i_c(c_j) = \top$ et que $\forall c_k \in F \setminus c,\ i_c(c_k) = \bot$. Si $(i_c \wedge mut = \top), \forall mut \in M$ (abrégé en $c \models M$ par la suite), alors c est une configuration partielle. On dit que c est maximale si $c \models IS \wedge C$.

Prenons l'exemple du configurateur de `SPLOT` [2], sur lequel on peut sélectionner pas-à-pas un ensemble de caractéristiques d'un modèle donné et qui complète cette sélection à la volée en fonction des contraintes du modèle. Par exemple, si l'on sélectionne f_1 et qu'il existe une contrainte $f_1 \rightarrow f_2$, le configurateur ajoute automatiquement f_2 à la configuration partielle courante. Les caractéristiques ajoutées automatiquement par le configurateur rendent la configuration maximale.

Ces trois types de contraintes étant par construction respectés par tous les concepts, les configurations partielles qu'ils représentent sont donc maximales. De plus, le treillis des concepts représentant toutes les paires d'ensembles maximaux du contexte formel, chaque configuration partielle qui peut être inférée automatiquement grâce aux contraintes inhérentes à la famille de logiciels correspond donc à l'intension d'un concept du treillis. En d'autres termes, chaque choix menant à une configuration valide et pouvant être réalisé par un utilisateur correspond à un concept du treillis.

Propriété 3.4. *L'ensemble des concepts n'introduisant pas de configurations représente toutes les configurations partielles maximales de la famille de logiciels.*

Démonstration. Soit $K = (O, A, I)$ un contexte formel et C_K l'ensemble des concepts formels associé à K. Soit IS, C et M trois ensembles représentant respectivement les

2. http ://splot-research.org/

implications binaires, les cooccurrences et les exclusions mutuelles vraies sur A. ($\forall c \subseteq A | c \models M$) alors $\exists C = (X, Y) \in C_K$ tel que ($Y = \beta(c), Y \models IS$ et $Y \models C$). ⌟

Exemple. Nous avons vu dans un exemple précédent que l'ensemble {`Operating System`, `Data Model`, `OS:Windows`, `OS:Mac OS`} représentait une configuration partielle. Elle n'est cependant pas maximale : en effet, les caractéristiques `OS:Linux` et `OS:Mac OS` apparaissent toujours ensemble dans les configurations valides et sont donc cooccurrentes. Si l'on rajoute `OS:Linux` à la combinaison initiale, les implications, cooccurrences et exclusions mutuelles sont alors toutes respectées. {`Operating System`, `Data Model`, `OS:Windows`, `OS:Mac OS`, `OS:Linux`} est donc une configuration partielle maximale. Elle correspond d'ailleurs au concept `Concept_DM_tools_6` du treillis de la Figure 3.2.

Si la combinaison de caractéristiques représentée par l'intension d'un concept est partagée par une seule configuration (i.e. l'extension du concept est de taille 1), alors le concept est un introducteur et cette combinaison est nécessairement une configuration valide, correspondant à l'extension. Si plusieurs configurations partagent cette combinaison (i.e. l'extension du concept est de taille ≥ 2), elle est commune à ces configurations et il existe plusieurs séquences d'ajout de caractéristiques possibles donnant une configuration valide. Puisque les contraintes pouvant être appliquées automatiquement sont naturellement respectées par les concepts, les séquences d'ajout dépendent donc uniquement des choix de l'utilisateur. Par conséquent, ces concepts peuvent être vus comme des nœuds de décision ; le treillis des concepts représentant toutes les configurations partielles maximales, il représente donc toutes les décisions pouvant être faites par l'utilisateur. Godin et al. définissent ces décisions comme minimales [GSG86], c'est-à-dire les plus petites décisions (en termes de modification sur l'ensemble des caractéristiques) pouvant être faites par l'utilisateur.

Propriété 3.5. *Chaque concept caractérise un nœud de décision à la charge de l'utilisateur durant le processus de sélection d'une configuration (i.e. entre plusieurs séquences d'ajout ou de suppression). Le treillis de concepts regroupe toutes les décisions minimales pouvant être faites par l'utilisateur.*

Les concepts retenus dans l'AOC-poset caractérisent aussi des décisions de l'utilisateur. Ces décisions ne sont cependant plus minimales, car un AOC-poset ne représente pas tous les concepts (les concepts neutres sont omis). Un concept neutre n'introduit ni configuration, ni caractéristique : son intension est donc égale à l'union des intensions de ses super-concepts.

Définition 3.13 (Concept neutre)**.** *Soit un contexte formel $K = (O, A, I)$. $C = (X, Y) \in C_K$ est un concept neutre $\iff$ C n'introduit pas de configuration et soient les concepts $C_i = (X_i, Y_i) \in C_K, i \in \{1, 2, \ldots, n\}$ et $C \leq_s C_i, Y = \bigcup_{i=1}^{n} Y_i$.*

Les nœuds de décision correspondant aux concepts neutres "enlevés" de l'AOC-poset peuvent être caractérisés comme des nœuds de décisions "intermédiaires" pouvant être fusionnés avec d'autres nœuds sans perdre d'information. En effet, d'après la définition précédente, la décision représentée par un concept neutre en termes de suppression de caractéristiques amène à l'un des nœuds de décision suivants : $C_1, \ldots, C_n$. Omettre ces nœuds de décision intermédiaires revient à combiner les décisions / séquences de suppression menant à C avec les décisions / séquences de suppression menant à $C_1, \ldots, C_n$. La même logique peut être appliquée de manière inverse aux séquences d'ajout. L'AOC-poset condense donc les séquences d'ajout et de suppression pouvant être appliquées sur une configuration partielle. Par exemple, imaginons qu'il existe deux séquences d'ajouts possibles applicables sur une configuration : f_1 puis f_2, ou bien f_1 puis f_3. Le treillis offre dans un premier temps la possibilité de sélectionner f_1, puis f_2 ou f_3 : la sélection requiert donc

deux décisions. L'AOC-poset quant à lui "factorise" ces séquences d'ajout possible en une seule décision $\{f_1, f_2\}$ ou $\{f_1, f_3\}$.

La place des concepts les uns par rapport aux autres donne des informations supplémentaires quant à ces décisions.

3.3.2 La place des concepts les uns par rapport aux autres

On appelle *voisinage conceptuel* d'un concept formel l'union de l'ensemble de ses super-concepts directs et de l'ensemble de ses sous-concepts directs dans le treillis ou les sous-ordres. Le voisinage conceptuel d'un concept représente les groupes maximaux de configurations les plus proches de l'extension de celui-ci. De plus, tous ces concepts sont comparables.

On appelle *couverture supérieure* d'un concept l'ensemble de ses super-concepts directs. L'extension du concept courant est strictement incluse dans chaque extension d'un concept de sa couverture supérieure. L'ensemble des caractéristiques de leurs intensions est donc strictement inclus dans l'intension du concept courant. Les super-concepts d'un concept sont donc obtenus par suppression de caractéristiques : dans le treillis, ils délimitent l'ensemble des configurations partielles maximales plus générales que la configuration étudiée. Les concepts de la couverture supérieure représentent donc la plus petite séquence de suppressions de caractéristiques respectant les contraintes données par les implications binaires.

Propriété 3.6. *Les extensions des concepts de la couverture supérieure d'un concept (i.e. nœuds de décision) du treillis des concepts représentent les décisions possibles de l'utilisateur en termes de suppression de caractéristiques qui modifient le moins l'ensemble courant des configurations.*

Exemple. Prenons l'exemple du `Concept_DM_tools_4` de la Figure 3.2. Son extension comprend les configurations correspondant aux variantes *Astah* et *Magic Draw*. Sa couverture supérieure est composée des deux concepts `Concept_DM_tools_6` et `Concept_DM_tools_8`. `Concept_DM_tools_6` représente la suppression des caractéristiques `OS:Mac OS` et `Linux` de l'ensemble des caractéristiques courant, ainsi que l'ajout de la configuration *MySQL Workbench* à l'ensemble {*Astah*, *Magic Draw*}. `Concept_DM_tools_8` représente la suppression de la caractéristique `DM:Conceptual` et l'ajout des configurations *Erwin DM* et *ER/Studio* à l'ensemble {*Astah*, *Magic Draw*}. Un concept placé plus haut dans le treillis impacte plus fortement l'ensemble courant des configurations : par exemple, le `Concept_DM_tools_9` ajoute les 3 configurations *MySQL Workbench*, *Erwin DM* et *ER/Studio* à l'ensemble {*Astah*, *Magic Draw*}.

De manière duale, on appelle *couverture inférieure* d'un concept l'ensemble de ses sous-concepts directs. La couverture inférieure possède les mêmes propriétés que la supérieure, mais cette fois-ci elles sont inversées. L'intension du concept courant est incluse dans chaque intension de la couverture inférieure et les extensions des concepts de la couverture inférieure sont incluses dans celle du concept courant. Les sous-concepts d'un concept sont donc obtenus par ajout de caractéristiques et délimitent donc l'ensemble des configurations partielles maximales plus spécifiques dans le treillis. Les sous-concepts directs représentent donc la plus petite séquence d'ajout de caractéristiques respectant les contraintes d'implication.

Propriété 3.7. *Les extensions des concepts de la couverture inférieure d'un concept (i.e. nœuds de décision) du treillis de concepts représentent les décisions possibles de l'utilisateur en termes d'ajout de caractéristiques qui modifient le moins l'ensemble courant des configurations.*

Exemple. Si l'on considère encore une fois le `Concept_DM_tools_4` de la Figure 3.2, on voit que sa couverture inférieure n'est constituée que du `Concept_DM_tools_1`. Le `Concept_DM_tools_1` correspond à l'ajout de la caractéristique `DM:Logical` : l'ensemble courant des configurations perd la configuration *Astah* et se réduit alors à la configuration *Magic Draw*.

On peut évaluer la similarité entre deux concepts comparables en fonction de leur place l'un par rapport à l'autre dans le treillis et l'AOC-poset. Soit trois concepts distincts et comparables $C_1 = (X_1, Y_1)$, $C_2 = (X_2, Y_2)$ et $C_3 = (X_3, Y_3)$. Si $C_1 <_s C_2 <_s C_3$, alors on aura dans le diagramme de Hasse un lien de C_1 vers C_2 et de C_2 vers C_3. De plus, puisque $X_1 \subset X_2 \subset X_3$, la différence entre X_1 est X_2 sera moins importante que celle entre X_1 et X_3.

Propriété 3.8. *Plus un concept est proche d'un autre concept avec lequel il est comparable dans le diagramme de Hasse du treillis ou de l'AOC-poset, plus leurs extensions sont similaires.*

Nous avons défini les super-concepts (resp. les sous-concepts) comme les concepts délimitant les configurations partielles maximales obtenues par application de séquences de suppression (resp. ajout) de caractéristiques. Les concepts non comparables à un concept donné sont donc obtenus par la composition d'au moins une séquence d'ajouts et d'une séquence de suppressions de caractéristiques. Plus ces séquences sont longues (i.e., la distance d'édition est importante) et plus le concept atteint est susceptible d'être éloigné du concept initial dans le diagramme de Hasse du treillis ou de l'AOC-poset et plus l'intersection de leurs intensions est susceptible d'être réduite. Ce n'est cependant pas vrai tout le temps, car la taille des séquences d'ajout ou de suppression de caractéristiques peut varier.

3.3.3 La place des concepts dans les structures

L'ordre partiel place les concepts ayant les extensions les plus grandes en haut du treillis. La dualité des structures conceptuelles de l'AFC fait qu'à l'inverse, les intensions les plus grandes sont en bas et les intensions les plus petites sont en haut. La place d'un concept dans le treillis permet donc d'estimer certaines propriétés sur les éléments qu'il introduit.

Deux concepts ayant une place particulière dans le treillis sont intéressants dans l'étude de la variabilité : le majorant et le minorant du treillis.

Le concept le plus haut dans le treillis, appelé *top-concept* est le majorant de tous les concepts du treillis. Ce concept introduit des caractéristiques si et seulement si celles-ci sont présentes dans toutes les configurations de la famille de logiciels étudiée, car elles sont héritées par toutes les configurations.

Propriété 3.9. *L'ensemble des caractéristiques communes à toutes les configurations (aussi appelées core-features) sont introduites dans le top-concept.*

Ce concept peut ne pas exister dans l'AOC-poset et dans l'AC-poset, auquel cas il n'y a pas de *core-features* dans la famille de logiciels.

Le concept le plus bas du treillis, appelé *bottom-concept* est le minorant de tous les concepts. S'il introduit des caractéristiques, mais que son extension est vide, cela signifie que ces caractéristiques ne sont présentes dans aucune des configurations valides.

Propriété 3.10. *S'il existe des caractéristiques absentes de toutes les configurations (aussi appelées dead-features), elles sont introduites dans le bottom-concept dont l'extension sera vide.*

Si le *bottom-concept* est un introducteur de caractéristiques mais que son extension n'est pas vide, cela signifie qu'il existe une configuration possédant toutes les caractéristiques de la FPL. De plus, on peut en conclure qu'il n'existe pas de paire de caractéristiques mutuellement exclusives. Ce concept peut ne pas exister dans l'AOC-poset et dans l'AC-poset, auquel cas il n'y a pas de *dead-features* dans la famille de logiciels.

On peut aussi tirer des conclusions en observant la place des autres concepts dans le treillis. Les concepts introducteurs de caractéristiques étant placés haut dans la structure ont plus de sous-concepts : les caractéristiques introduites sont donc partagées par plus de configurations et sont donc plus communes. À l'inverse, les caractéristiques introduites dans des concepts plus bas dans le treillis sont susceptibles d'être partagées par moins de configurations et peuvent être considérées comme plus rares, ou plus spécifiques.

Propriété 3.11. *Plus une caractéristique est introduite haut dans le treillis, plus le nombre de configurations qui la partagent est susceptible d'être grand.*

À l'inverse, les configurations introduites dans des concepts en haut du treillis peuvent être considérées comme des configurations plus générales. En effet, elles sont susceptibles de posséder moins de caractéristiques que les configurations introduites plus bas, qui sont donc généralement plus spécifiques. De plus, ces configurations plus générales vont posséder des caractéristiques plus communes, car introduites plus haut.

Propriété 3.12. *Plus une configuration est introduite haut dans le treillis, plus le nombre de caractéristiques qui la décrivent est susceptible d'être petit.*

La place des concepts introducteurs les uns par rapport aux autres donne des informations sur la spécialisation / généralisation des éléments qu'ils introduisent. Si une configuration c_1 est introduite dans un sous-concept d'un autre concept introduisant une configuration c_2, alors c_1 possède toutes les caractéristiques de c_2, ainsi que d'autres caractéristiques : la configuration c_1 est donc plus spécifique que c_2. À l'inverse, si une caractéristique a_1 est introduite dans un super-concept d'un autre concept introduisant la caractéristique a_2, alors toutes les configurations ayant a_2 ont aussi a_1. Ainsi, on peut observer des relations entre caractéristiques qui sont vraies dans l'ensemble de configurations étudié, par exemple, l'implication $a_2 \rightarrow a_1$. Ceci fait l'objet de la section suivante.

En résumé (3.3) :

- Les **concepts formels** donnent des informations sur la **répartition des caractéristiques dans les configurations valides**.
- Exception faite du *bottom-concept*, les concepts représentent toutes les **configurations partielles maximales** de la famille de logiciels, ainsi que les **nœuds de décisions** soumis à l'utilisateur pendant la sélection.
- Le **voisinage conceptuel** détermine les concepts **les plus similaires** du concept courant, ainsi que les **décisions minimales** pouvant être prises par l'utilisateur pour modifier une configuration partielle dite maximale.
- L'**AOC-poset** présente un sous ensemble **condensé** de ces décisions.
- Le majorant de tous les concepts et le minorant de tous les concepts présentent respectivement les ***core-features*** et les ***dead-features***.
- Le **caractère spécifique** d'une caractéristique ou d'une configuration peut être évalué à travers sa **place dans la structure**.

3.4 Sur l'extraction de variabilité depuis les structures conceptuelles de l'AFC

Dans la section précédente, nous avons vu les différentes informations sur la variabilité qui sont inhérentes aux structures conceptuelles. Dans cette section, nous montrons comment utiliser ces propriétés de l'AFC pour extraire différents types de relations logiques depuis ces structures. Plus précisément, nous étudions les relations logiques correspondant à la sémantique logique des FMs, telle que nous l'avons présentée en Section 2.4, i.e. les implications binaires (Section 3.4.1), les cooccurrences (Section 3.4.2), les mutex (Section 3.4.3) et les groupes *or* et *xor* (Section 3.4.4). Pour chacune de ces relations, nous discutons des bases mathématiques de son extraction, donnons un algorithme et montrons pourquoi la méthode d'extraction proposée est valide et complète. Nous réalisons ensuite une étude comparative des différentes approches d'extraction de variabilité que l'on trouve dans la littérature (Section 3.4.5)

3.4.1 Extraire les implications binaires

Les implications binaires sont la traduction en logique des propositions des relations parent-enfant, des relations optionnelles et des implications binaires transverses que l'on trouve dans les FMs.

L'ensemble des implications binaires entre les attributs d'un contexte formel peuvent être lues dans le treillis de concepts qui lui est associé. Plus formellement, Ganter et Wille [GW99, p.80] exhibent la propriété suivante :

Propriété 3.13. *Une implication binaire $a_1 \rightarrow a_2$ est vraie dans un contexte formel $K = (O, A, I)$ si et seulement si $a_2 \in \alpha \circ \beta(\{a_1\})$, avec $a_1, a_2 \in A$.*

En effet, $\alpha \circ \beta(\{a_1\})$ donne "l'ensemble de tous les attributs communs à l'ensemble des objets possédant l'attribut a_1" : si a_2 est dans cet ensemble d'attributs, alors cela signifie que tous les objets ayant l'attribut a_1 ont aussi l'attribut a_2. En d'autres termes, si le concept $C_1 = (X_1 = \beta(\{a_1\}), Y_1 = \alpha \circ \beta(\{a_1\}))$ introduisant l'attribut a_1 vérifie $a_2 \in Y_1$, alors a_2 est soit introduit dans le concept C_1, soit introduit dans un super-concept de C_1. Ceci est dû au fait que dans un treillis de concepts, un concept hérite de l'ensemble des attributs de ses super-concepts. On peut donc reformuler la propriété précédente par :

Propriété 3.14. *Soit C_1 et C_2 deux concepts de K introduisant respectivement les attributs a_1 et a_2, $C_1 \leq_s C_2 \iff a_1 \rightarrow a_2$.*

On peut tirer deux conclusions de ces propriétés :

1. Toutes les implications binaires qui sont vraies dans un contexte formel peuvent être extraites de son treillis de concepts associé : l'extraction d'implications binaires depuis un treillis de concepts est donc *complète*.
2. Toutes les implications que l'on peut extraire d'un treillis de concepts sont vraies dans son contexte formel associé : l'extraction d'implications binaires depuis un treillis de concepts est donc *valide*.

Il y a une équivalence entre un contexte formel, le treillis de concept qui lui est associé ainsi que l'ensemble des implications binaires qui peuvent en être extraites. Il existe toute une littérature sur la relation entre les implications binaires et l'AFC [Ber11]. En particulier, il a été démontré que l'ensemble des implications binaires (sur les attributs) d'un contexte formel permet de construire un treillis des fermés sur les attributs qui est isomorphe au treillis de concepts associé avec le contexte formel initial.

Algorithme 1 : ExtractAllBinaryImplications

Data : C_K : set of formal concepts of a formal context K, S : the set of all ordered pairs of concepts (C_i, C_j) from C_K such that $C_i \leq_s C_j$, representing the partial order between the formal concepts of K

Result : Imp : set of all binary implications that hold in K

```
1 foreach (C_i, C_j) ∈ S do
2     foreach a_i ∈ SimplifiedIntent(C_i) do
3         foreach a_j ∈ SimplifiedIntent(C_j) do
4             Imp ← Imp ∪ {a_i → a_j}
```

L'Algorithme 1 définit un processus d'extraction des implications binaires depuis un treillis de concepts.

En entrée, cet algorithme prend le treillis de concepts associé à un contexte formel K, représenté par 1) l'ensemble $C_K = \{C_i\}$ des concepts de K et 2) la relation d'ordre $\leq_s$ entre ces concepts, exprimée par l'ensemble $S \subseteq C_K \times C_K$ de tous les couples $(C_i, C_j) \in S$ tels que $C_i \leq_s C_j$. En sortie, il donne l'ensemble des implications binaires qui sont vraies dans K. Pour cela, l'algorithme va parcourir l'ensemble des couples (C_i, C_j) de S (ligne 1). La fonction `SimplifiedIntent(C)` avec $C \in C_k$ retourne l'intension simplifiée du concept C, i.e. l'ensemble des attributs qui sont introduits dans C. Ensuite, pour chaque attribut a_i introduit dans C_i et pour chaque attribut a_j introduit dans C_j, on ajoute dans l'ensemble Imp l'implication $a_i \rightarrow a_j$ (lignes 2-4).

Soit n le nombre de concepts de C_K et m la taille de l'ordre partiel entre les concepts de C_K bornée supérieurement par $m \leq \frac{n(n+1)}{2}$. Le nombre d'implications binaires extraites $|Imp|$ est borné supérieurement par $|Imp| \leq |A|^2$. Dans le pire des cas, et si la structure conceptuelle est déjà construite, la complexité de l'Algorithme 1 est donc dans $\mathcal{O}(m+|A|^2)$.

Notons qu'il n'est pas nécessaire de considérer l'intégralité du treillis de concepts pour extraire l'ensemble de toutes les implications binaires. En effet, leur identification nécessite les concepts introducteurs d'attributs et l'ordre partiel restreint à ces concepts. On peut donc extraire l'ensemble des implications binaires depuis l'AOC-poset et depuis l'AC-poset. L'AC-poset représente un graphe d'implications binaires entre les attributs du contexte formel ; ce graphe d'implications est donc inclus dans l'AOC-poset et le treillis de concepts.

Dans l'algorithme donné ci-dessus, on parcourt tous les couples de concepts (C_i, C_j) de S sans savoir si C_i et C_j introduisent des attributs, i.e. si leur intension simplifiée est non vide. On peut alors réduire la complexité de l'Algorithme 1 en ne prenant en entrée que les concepts de C_K introduisant des attributs, i.e. l'ensemble des concepts introducteurs d'attributs AC_K et en réduisant l'ensemble S aux couples de concepts appartenant tous deux à AC_K. L'algorithme reste le même, mais la complexité dans le pire des cas change. Le nombre de concepts introducteurs d'attributs est borné supérieurement par le nombre d'attributs (car chaque attribut est introduit exactement une fois) : $|AC_K| \leq |A|$. La taille m' du sous-ordre restreint aux concepts introducteurs d'attributs est donc bornée supérieurement par $m' \leq \frac{|A|(|A|+1)}{2}$. Sachant que $n \leq 2^{|A|}$, en partant de l'AC-poset on obtient une meilleure complexité dans $\mathcal{O}(m' + |A|^2)$.

Puisque cette méthode permet d'extraire toutes les implications binaires vraies dans K, il est possible que certaines d'entre elles soient redondantes. Cela signifie qu'elles peuvent être retrouvées à partir des autres implications binaires par transitivité de la règle de déduction : si $a \rightarrow b$ et $b \rightarrow c$, alors $a \rightarrow c$ est vraie mais peut être omise sans perte d'information. Le graphe d'implications binaires associé à toutes les implications binaires équivaut donc à sa fermeture transitive, qui est un abus de langage pour parler de la transitivité de la règle de déduction. Il est cependant possible d'extraire directement la

réduction transitive de l'ensemble des implications binaires depuis le treillis de concepts, l'AOC-poset ou l'AC-poset. Nous avons vu que ces trois structures conceptuelles incluent le graphe d'implications binaires sur les attributs. L'AC-poset est équivalent à ce graphe : si l'on ne garde en entrée de l'Algorithme 1 que les couples concept/super-concept directs dans S, correspondant à la réduction transitive de l'AC-poset, on extrait alors uniquement les implications de la réduction transitive. Depuis le treillis et l'AOC-poset, le processus est plus compliqué : pour chaque concept introduisant un attribut, il faut chercher ses super-concepts introduisant au moins un attribut et qui soient de plus les plus bas dans la structure. Ces derniers correspondent aux super-concepts directs dans l'AC-poset associé.

On peut donc extraire la réduction transitive en modifiant l'entrée de l'Algorithme 1 ; cette fois-ci on considère toujours l'ensemble de concepts introducteurs d'attributs AC_K, mais on réduit encore S pour prendre en compte les couples (C_i, C_j) tels que $C_i, C_j \in AC_K$, $C_i \leq_s C_j$ et $\nexists C_l \in AC_K | C_i \leq_s C_l$ et $C_l \leq_s C_j$. Dans le pire des cas, la réduction transitive des implications binaires est égale à la fermeture transitive, on a donc une complexité dans $\mathcal{O}(m' + |A|^2)$.

Exemple. Dans le treillis de concepts présenté en Figure 3.2 (page 38), on peut par exemple lire les implications binaires suivantes :

1. `DM:ETL` $\rightarrow$ `DM:Logical` (car `Concept_DM_tools_2` $\leq_s$ `Concept_DM_tools_5`)
2. `DM:ETL` $\rightarrow$ `DM:Conceptual` (car `Concept_DM_tools_2` $\leq_s$ `Concept_DM_tools_8`)
3. `DM:Logical` $\rightarrow$ `OS:Windows` (car `Concept_DM_tools_5` $\leq_s$ `Concept_DM_tools_9`)
4. `OS:Mac` $\rightarrow$ `OS:Linux` (car `Concept_DM_tools_6` $\leq_s$ `Concept_DM_tools_6`)
5. `OS:Linux` $\rightarrow$ `OS:Mac` (car `Concept_DM_tools_6` $\leq_s$ `Concept_DM_tools_6`)

Notons que cette liste n'est pas exhaustive et que les implications 1, 4 et 5 font partie de la réduction transitive.

3.4.2 Extraire les cooccurrences

Les équivalences sont la traduction en logique des propositions des cas où des paires de caractéristiques apparaissent toujours ensemble dans toutes les configurations valides d'un FM, i.e. les relations *obligatoires* et les doubles implications transverses. On dit que ces deux caractéristiques sont cooccurrentes.

C'est un cas particulier des implications binaires, où l'attribut a_1 implique a_2 et où l'attribut a_2 implique a_1 en retour. On peut exprimer ce cas particulier dans les termes de la Propriété 3.13 précédente :

Propriété 3.15. *Les deux implications binaires $a_1 \rightarrow a_2$ et $a_2 \rightarrow a_1$ (notées $a_1 \leftrightarrow a_2$) sont vraies dans un contexte formel $K = (O, A, I)$ si et seulement si $a_2 \in \alpha \circ \beta(\{a_1\})$ et $a_1 \in \alpha \circ \beta(\{a_2\})$.*

En d'autres termes, si les concepts $C_1 = (X_1 = \beta(\{a_1\}), Y_1 = \alpha \circ \beta(\{a_1\}))$ et $C_2 = (X_2 = \beta(\{a_2\}), Y_2 = \alpha \circ \beta(\{a_2\}))$ vérifient $a_1 \in Y_2$ et $a_2 \in Y_1$, alors $C_1 = C_2$ et a_1 et a_2 sont donc introduits dans le même concept. Les cooccurrences peuvent donc être lues dans le treillis : ce sont les ensembles d'attributs qui sont introduits dans le même concept. Dans le contexte formel, ce sont les attributs qui ont des colonnes identiques. L'ensemble des cooccurrences peut être extrait des structures conceptuelles incluant l'ensemble des concepts introducteurs d'attributs : en plus du treillis de concepts, on peut donc utiliser l'AOC-poset et l'AC-poset. Étant basée sur la même propriété que celle vue précédemment avec l'extraction des implications binaires, cette méthode d'extraction est donc elle aussi *complète* et *valide*.

Algorithme 2 : ExtractAllCoOccurrences

Data : AC_K : set of attribute-concepts of a formal context K
Result : $Cooc$: set of all cooccurrences that hold in K
1 **foreach** $C \in AC_K$ **do**
2 **if** $|SimplifiedIntent(C)| > 1$ **then**
3 **foreach** *pair* $(a_i, a_j) \in SimplifiedIntent(C)$ **do**
4 $Cooc \leftarrow Cooc \cup \{a_i \leftrightarrow a_j\}$

L'Algorithme 2 définit le processus d'extraction des cooccurrences depuis un treillis de concepts, un AOC-poset ou un AC-poset.

En entrée, il prend l'ensemble des concepts introducteurs d'attributs AC_K d'un contexte formel et il donne en sortie l'ensemble des cooccurrences qui sont vraies dans ce contexte. Pour cela, il parcourt chaque concept de AC_K : si le concept introduit plus d'un attribut, alors les attributs de chaque paire d'attributs introduits dans ce concept sont cooccurrents : ils sont gardés sous forme d'équivalences logiques. Le nombre de concepts introducteurs d'attributs est borné supérieurement par le nombre d'attributs $|A|$ et il y a au plus $\frac{|A|(|A|-1)}{2}$ cooccurrences entre les attributs d'un contexte formel ; la complexité dans le pire des cas est donc de l'ordre de $\mathcal{O}(|A|^2)$.

Exemple. Dans le treillis de concepts présenté en Figure 3.2 (page 38), on n'observe qu'une seule cooccurrence dans le `Concept_DM_tools_6` : `OS:Mac` $\leftrightarrow$ `OS:Linux`.

3.4.3 Extraire les mutex

En logique des propositions, on représente l'exclusion mutuelle (mutex) entre deux caractéristiques a_1 et a_2 par l'expression $a_1 \rightarrow \neg a_2$ (ou son équivalent $a_2 \rightarrow \neg a_1$) : c'est ainsi que sont représentés les mutex transverses dans les FMs.

Si deux attributs d'un contexte formel sont mutuellement exclusifs, cela signifie qu'ils n'apparaissent jamais ensemble dans la description d'un objet et ce pour tous les objets du contexte. C'est-à-dire que l'ensemble des objets possédant le premier attribut et l'ensemble des objets possédant le second attribut sont disjoints. On peut donc naturellement formuler la propriété suivante :

Propriété 3.16. *L'exclusion mutuelle $a_1 \rightarrow \neg a_2$ est vraie dans un contexte formel $K = (O, A, I)$ si et seulement si $\beta(\{a_1\}) \cap \beta(\{a_2\}) = \varnothing$.*

En d'autres termes, si les concepts $C_1 = (X_1 = \beta(\{a_1\}), Y_1 = \alpha \circ \beta(\{a_1\}))$ et $C_2 = (X_2 = \beta(\{a_2\}), Y_2 = \alpha \circ \beta(\{a_2\}))$ (introduisant respectivement les attributs a_1 et a_2) vérifient $X_1 \cap X_2 = \varnothing$, alors a_1 et a_2 sont mutuellement exclusifs. Cette méthode d'identification est complète et valide.

Cette information peut elle aussi être lue dans le treillis de concepts. En effet, la composition $\alpha \circ \beta$ est un opérateur de fermeture sur l'ensemble des parties d'attributs du contexte formel, qui est stable par intersection $\cap$ [BM70]. Cela signifie que l'intersection des extensions de deux concepts du treillis correspond toujours à l'extension d'un concept dans le treillis. On a donc la propriété suivante :

Propriété 3.17. *Soit K un contexte formel et C_K l'ensemble des concepts de K. $\forall C_1, C_2 \in C_K$ avec $C_1 = (X_1, Y_1)$ et $C_2 = (X_2, Y_2)$, alors $\exists C_3 = (X_3, Y_3)$ tel que $C_3 \in C_K$, $X_1 \cap X_2 = X_3$ et $Y_3 = \alpha(X_3)$.*

Le concept représentant cette intersection est la borne inférieure (ou le plus grand minorant) des deux concepts dans le treillis, qui existe par définition. Donc, si le plus

grand minorant de deux concepts possède une extension vide, alors les attributs introduits dans les deux concepts de départ sont mutuellement exclusifs. Dans un treillis de concepts, un seul concept peut posséder une extension vide : le *bottom-concept*. Le *bottom-concept* est le concept le plus bas du treillis (i.e. il n'a pas de minorant autre que lui même). C'est donc le concept qui hérite de tous les attributs du contexte formel. Si le *bottom-concept* a une extension non vide, cela signifie qu'il existe au moins un objet possédant tous les attributs du contexte formel et donc qu'il n'existe pas d'exclusions mutuelles. Cela représente un test rapide permettant de savoir s'il est nécessaire de chercher des mutex dans le contexte formel. S'il existe des exclusions mutuelles, alors l'extension du *bottom-concept* est vide et il est le minorant de toutes les paires de concepts introduisant des attributs mutuellement exclusifs. L'extension peut être vide, sans qu'il existe de mutex : ce test permet donc de savoir s'il est nécessaire d'en chercher, pas de confirmer qu'il en existe.

La Figure 3.5 montre un exemple de treillis de concepts dans lequel il n'y a pas de paires d'attributs mutuellement exclusifs, mais pour lequel le *bottom-concept* est vide. Ici, on ne trouve jamais les 4 attributs ensemble, mais on peut trouver tous les couples et tous les triplets. Il existe donc un ensemble d'attributs de taille > 2 que l'on ne trouve jamais ensemble ; ces relations "*not all together*" ne sont cependant pas étudiées ici car elles ne sont pas représentées dans les FMs.

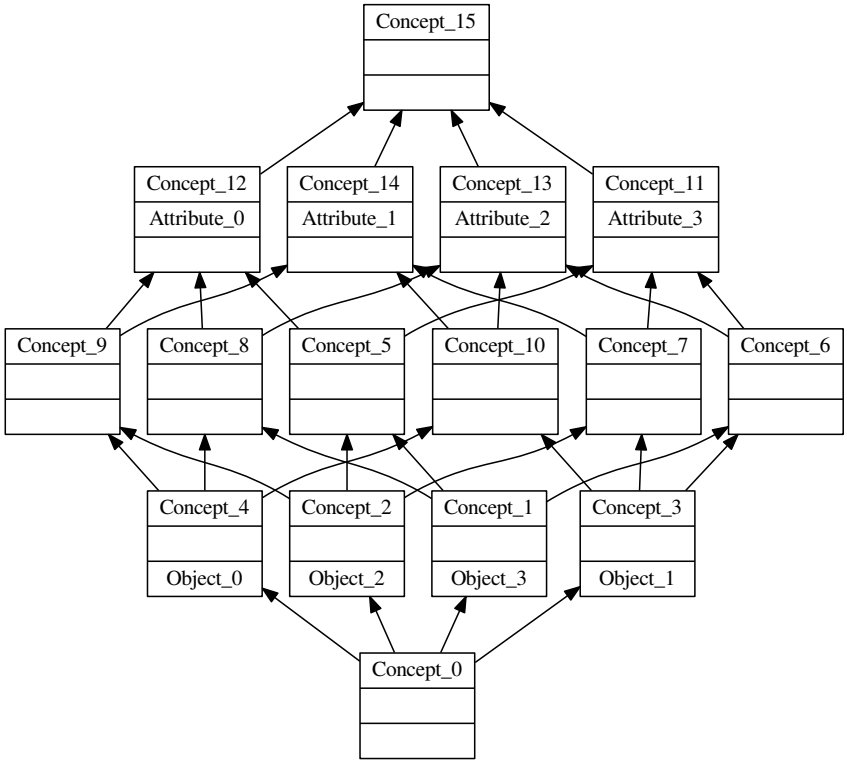


FIGURE 3.5 – Exemple de treillis de concepts dans lequel le *bottom-concept* possède une extension vide, mais où il n'y pas de paires d'attributs mutuellement exclusifs

Là encore, il est suffisant de travailler sur les concepts introducteurs d'attributs. Cependant, il se peut que le *bottom-concept* ne soit pas représenté dans les sous-ordres du treillis de concepts. Si son extension est vide (i.e. il n'y a pas d'objet possédant tous les attributs du contexte) et que son intension est vide aussi (i.e. il n'y a pas d'attributs qui ne sont possédés par aucun objet du contexte), alors c'est un concept neutre qui n'apparaît pas dans les sous-hiérarchies telles que l'AOC-poset et l'AC-poset. Dans l'AOC-poset, si deux concepts introducteurs d'attributs n'ont pas de minorant, alors l'intersection de leurs

extensions est vide. Ils peuvent cependant avoir un minorant dont l'extension est vide (par exemple s'il existe un attribut qui n'est partagé par aucun objet, il sera introduit dans ce concept). Tester si deux concepts ont un minorant dans l'AOC-poset n'est donc pas suffisant pour savoir s'ils sont mutuellement exclusifs : s'il existe un minorant, il faut vérifier son extension, comme dans le treillis de concepts. Cela ne fonctionne pas pour l'AC-poset, qui ne montre que les introducteurs d'attributs : deux concepts peuvent ne pas avoir de minorant, mais posséder des objets communs dans leurs extensions. Il faut donc calculer l'intersection des extensions de ces deux concepts.

Par la suite, même si le calcul du plus grand minorant et l'analyse de son extension permettent d'observer les exclusions mutuelles dans les structures conceptuelles, nous choisissons la solution qui consiste à calculer l'intersection des extensions des concepts introducteurs d'attributs, plus facile à mettre en œuvre et applicable de manière générique aux trois structures conceptuelles supportant l'extraction des mutex.

L'Algorithme 3 définit le processus d'extraction des mutex depuis un treillis de concepts, un AOC-poset ou un AC-poset.

Algorithme 3 : ExtractAllMutex

```
   Data : AC_K : set of attribute-concepts of a formal context K
   Result : Mutex : set of all mutex that hold in K
1  foreach i ∈ {0, 1, ..., |AC_K| − 2} do
2      foreach j ∈ {i + 1, i + 2, ..., |AC_K| − 1} do
3          if X_i ∩ X_j = ∅ then
4              foreach a_i ∈ SimplifiedIntent(C_i) do
5                  foreach a_j ∈ SimplifiedIntent(C_j) do
6                      Mutex ← Mutex ∪ {a_i → ¬a_j}
```

En entrée, il prend l'ensemble des concepts introducteurs d'attributs d'un contexte formel K et donne en sortie l'ensemble des mutex qui sont vrais dans K. Dans un premier temps, on parcourt exactement une fois toutes les paires de concepts (C_i, C_j) avec $i \neq j$ (lignes 1-2). On suppose les concepts introducteurs d'attributs numérotés de 0 à $|AC_K|-1$. Pour chaque paire, on teste si l'intersection de leurs extensions est vide (ligne 3). Si oui, alors les attributs introduits dans ces deux concepts sont mutuellement exclusifs : on parcourt toutes les paires d'attributs possibles parmi les attributs introduits dans les deux concepts (lignes 4-5), puis on ajoute le mutex correspondant (ligne 6).

Le nombre de paires de concepts introducteurs d'attributs (C_i, C_j) avec $i \neq j$ est de $\frac{|AC_K|(|AC_K|-1)}{2}$, avec $|AC_K| \leq |A|$. De même, le nombre de mutex d'un contexte formel est borné supérieurement par $\frac{|A|(|A|-1)}{2}$. Si les objets dans les extensions des concepts sont triés, l'intersection peut se faire en $\mathcal{O}(|X_i| + |X_j|)$, ce qui est borné supérieurement par $2|O|$. La complexité de l'algorithme est donc de l'ordre de $\mathcal{O}(|O||A|^2)$.

Exemple. Dans le treillis de concepts de la Figure 3.2 (page 38), les deux seuls concepts introducteurs d'attributs ayant des extensions dont l'intersection est vide sont le `Concept_DM_tools_2` (introduisant `DM:ETL`) et le `Concept_DM_tools_6` (introduisant `OS:Mac` et `OS:Linux`). On a donc les deux mutex suivants : `DM:ETL` $\rightarrow \neg$ `OS:Mac` et `DM:ETL` $\rightarrow \neg$ `OS:Linux`.

Il existe une propriété des treillis de concepts permettant de soulever deux remarques sur l'extraction des mutex : $\forall C_1 = (X_1, Y_1), C_2 = (X_2, Y_2) \in C_K, C_1 \leq_s C_2 \Rightarrow X_1 \subseteq X_2$.

Remarque 1 : *Il est possible de déduire certains mutex à partir d'autres mutex en utilisant les structures conceptuelles.*

En effet, si $X_1 \cap X_2 = \varnothing$ et que $\exists C_3 = (X_3, Y_3), C_4 = (X_4, Y_4) \in C_K$ tels que $C_3 \leq_s C_1$ et $C_4 \leq_s C_2$, alors $X_3 \cap X_4 = \varnothing$. Plus concrètement, si deux attributs introduits dans des concepts en haut du treillis sont mutuellement exclusifs, alors les attributs introduits dans des sous-concepts de ces deux introducteurs sont aussi mutuellement exclusifs. Grâce à l'ensemble des implications binaires d'un contexte formel, on peut dériver les mutex les plus spécialisés depuis des mutex plus généraux.

$$\forall a, b, c, d \in A, ((a \rightarrow \neg b) \wedge (c \rightarrow a) \wedge (d \rightarrow b)) \Rightarrow ((c \rightarrow \neg d) \wedge (c \rightarrow \neg b) \wedge (d \rightarrow \neg a)).$$

Dans l'algorithme présenté ci-dessus, nous choisissons d'extraire tous les mutex possibles, dans le cas où ils seraient étudiés séparément des implications binaires.

Remarque 2 : *Il est possible d'éliminer en avance certaines paires de concepts introducteurs d'objets dont l'intersection des extensions n'est pas vide.*

En effet, $\forall C_1 = (X_1, Y_1), C_2 = (X_2, Y_2) \in C_K, C_1 \leq_s C_2$ avec $C_1 \neq \perp$ et $C_2 \neq \perp \Rightarrow X_1 \cap X_2 \neq \varnothing$. En d'autres termes, si deux concepts introducteurs d'attributs (différents du *bottom-concept*) sont comparables, alors l'intersection de leurs extensions n'est jamais vide. La fermeture transitive de l'ordre partiel entre les concepts représente naturellement les paires de concepts comparables. Si l'on prend le complémentaire de son graphe d'implications binaires non-orienté, on obtient un graphe représentant l'ensemble des paires de concepts non comparables. Le graphe obtenu inclut le "graphe des mutex" présenté par She *et al.* [SLB+11, SRA+14]. Le graphe des mutex est un graphe dont les nœuds représentent les attributs et où une arête entre deux nœuds signifie qu'ils sont mutuellement exclusifs. Ce graphe des concepts introducteurs d'attributs incomparables donne les paires candidates à la recherche de mutex et réduit considérablement le nombre de paires à tester.

3.4.4 Extraire les groupes

Nous cherchons à présent à identifier dans le treillis des groupes d'attributs qui ont le même comportement que les groupes de caractéristiques dans les FMs. Un groupe est un ensemble de caractéristiques enraciné sur une caractéristique mère. Grouper des caractéristiques signifie que si la caractéristique mère est présente dans une configuration valide, au moins une des filles doit l'être aussi. Dans la notation FODA, il existe deux types de groupes : les groupes *or* et les groupes *xor*. Les groupes *or* ne contraignent pas le nombre de caractéristiques maximum qu'il faut sélectionner ([1..n]), alors que les groupes *xor* réduisent ce nombre à 1 ([1..1]). Les groupes *xor* sont donc un sous-cas des groupes *or*.

Les groupes *or* et *xor* possèdent donc les propriétés suivantes :

1. chaque caractéristique du groupe implique la même caractéristique mère ;
2. dans chaque configuration valide, la caractéristique mère doit toujours apparaître avec au moins une des caractéristiques du groupe ;
3. *chaque combinaison de caractéristiques autorisée par le groupe doit apparaître avec la caractéristique mère dans au moins une configuration.*

Si toutes les combinaisons de caractéristiques autorisées par la cardinalité du groupe existent dans l'ensemble des configurations étudié, on dit que le groupe *or* extrait est *pur*. Autrement dit, ces groupes respectent la propriété 3. Cependant, puisque nous travaillons sur des descriptions incomplètes, il est très improbable de détecter des groupes purs. Par la suite, nous chercherons donc à extraire des groupes qui répondent aux propriétés 1 et 2 : les groupes *or* étudiés par la suite sont sous-entendus non purs.

Les groupes *xor* vérifient en plus la contrainte suivante :

4. Dans chaque configuration valide, la caractéristique mère doit toujours apparaître avec au plus une des caractéristiques du groupe.

Par définition, les groupes *xor* extraits sont donc toujours purs.

Nous allons d'abord voir comment localiser les attributs ayant les deux comportements caractérisant les groupes *or* (non purs). Puis, nous verrons comment détecter le comportement plus spécifique des groupes *xor* parmi les groupes *or* obtenus.

Par la suite, nous désignerons par a_0 l'attribut parent d'un groupe et $C_0 = (X_0, Y_0)$ le concept formel de C_K introduisant a_0.

Extraire les groupes *or* (non purs)

Dans un premier temps, les attributs qui vont faire partie d'un groupe devront tous avoir le même parent. C'est-à-dire que tous les attributs du groupe vont devoir *impliquer* l'attribut parent (propriété 1). En fonction d'un attribut a_0 du contexte formel, on peut donc identifier les attributs candidats pour former un groupe sous a_0 dans le treillis. En effet, l'ensemble des attributs qui impliquent a_0 sont les attributs introduits dans les sous-concepts de C_0, l'introducteur de a_0. Les attributs candidats pour faire partie d'un groupe dont a_0 est le parent sont :

$$CandidatsGroupe(a_0) = \{a \in A | a_0 \in \alpha \circ \beta(\{a\})$$

Pour former un groupe, le nombre de candidats doit être supérieur ou égal à deux.

Exemple. Dans le treillis de la Figure 3.2 (page 38), si l'on prend l'exemple de l'attribut `Data Model` introduit dans le concept 9, l'ensemble des attributs candidats pour former un groupe sont ceux introduits dans les sous-concepts du concept 9. Cela correspond aux attributs introduits dans les concepts 2, 5, 6, 7 et 8, à savoir `OS:Mac OS`, `OS:Linux`, `DM:Physical`, `DM:Conceptual`, `DM:Logical` et `DM:ETL`.

Maintenant, si l'on prend l'exemple de l'attribut `DM:Conceptual` introduit dans le concept 8, les attributs candidats sont introduits dans les concepts 2 et 5, à savoir `DM:ETL` et `DM:Logical`.

Dans un second temps, un groupe doit respecter la propriété 2 stipulant qu'au moins un attribut du groupe apparaît tout le temps avec l'attribut parent a_0 dans le contexte formel. En d'autres termes, il ne doit pas exister d'objet dans O possédant a_0 mais aucun des attributs introduits sous C_0.

Propriété 3.18. *L'attribut a_0 peut être le parent d'un groupe si et seulement si $\forall o \in O | a_0 \in \alpha(\{o\}), \exists a \in A | a \in \alpha \circ \beta(\{a_0\})$ et $a \in \alpha(\{o\})$.*

Cette information peut être lue dans le treillis. En effet, pour que cette propriété soit vérifiée, il faut qu'il y ait au moins un concept introducteur d'attribut entre C_0 et chaque sous-concept de C_0 introduisant un objet. Si un introducteur d'objet C est sous-concept de C_0, mais qu'il n'y a pas de concept introducteur d'attribut qui soit sous-concept de C_0 et super-concept de C, cela signifie que C possède dans son intension a_0, mais aucun attribut introduit sous C_0 : la propriété 3.18 n'est donc pas respectée.

Exemple. Dans le treillis de la Figure 3.2 (page 38), le concept 8 introduisant l'attribut `DM:Conceptual` a pour sous-concept direct le concept 4, introduisant l'objet *Astah*. Cela signifie que *Astah* possède l'attribut `DM:Conceptual` mais aucun des attributs candidats pour former un groupe sous cette caractéristique (à savoir `DM:Logical` et `DM:ETL`). Les attributs introduits dans le concept 8 ne peuvent donc pas être des parents de groupe.

A contrario, le concept 9 possède trois sous-concepts directs qui sont des concepts introducteurs d'attributs : les attributs introduits dans le concept 9 peuvent donc être des parents de groupes.

Il est encore plus facile de vérifier cette propriété dans l'AC-poset. Si l'union des extensions des concepts formant le voisinage inférieur direct de C_0 ne couvre pas l'extension de C_0, alors il existe au moins un objet introduit entre C_0 et ces introducteurs d'attributs dans le treillis de concepts et l'AOC-poset associés. On peut donc reformuler la propriété 3.18 comme suit :

Propriété 3.19. *Soit l'attribut a_0 introduit dans le concept $C_0 = (X_0, Y_0) \in C_K$. Soient les concepts $C_i = (X_i, Y_i) \in AC_K, i \in \{1, 2, \ldots, n\}, i > 1 | C_i <_s C_0$ et $\nexists C_j \in C_K, (C_j <_s C_0$ et $C_i \leq_s C_j)$. a_0 peut être le parent d'un groupe si et seulement si $X_0 = \bigcup_{i=1}^{n} X_i$.*

Exemple. Étudions maintenant l'AC-poset à gauche de la Figure 3.4 (page 40). L'extension du concept 8 introduisant l'attribut `DM:Conceptual` est égale à {*Astah*, *Erwin DM*, *Magic Draw*, *ER/Studio*}. L'union des extensions des sous-concepts du concept 8 introduisant des attributs (i.e. les concepts 2 et 5) est égale à {*Erwin DM*, *Magic Draw*, *ER/Studio*}. Ces deux ensembles étant différents, on peut conclure que `DM:Conceptual` ne peut pas être un parent de groupes.

Nous venons donc de voir comment vérifier si un attribut a_0 pouvait être un parent de groupe et savoir quels attributs pouvaient être compris dans ce groupe. Autrement dit, nous savons maintenant détecter un attribut parent et un groupe d'attributs tels que l'attribut parent apparaît toujours avec au moins un de ces attributs du groupe dans le contexte formel : les groupes *or* (non purs).

Détecter les groupes *xor*

Un ensemble d'attributs et leur attribut parent forment un groupe *xor* si 1) c'est un groupe *or* et 2) si l'attribut parent apparaît toujours avec exactement un des attributs du groupe dans le contexte formel. Cela signifie qu'il n'existe pas de paires d'attributs appartenant au groupe qui apparaissent ensemble dans les descriptions des objets du contexte formel. En d'autres termes, un groupe *xor* est un groupe dans lequel tous les attributs sont mutuellement exclusifs.

Propriété 3.20. *Soit $\{C_1 = (X_1, Y_1), C_2 = (X_2, Y_2), \ldots, C_n = (X_n, Y_n)\}$ l'ensemble des concepts de K introduisant respectivement les attributs $\{a_1, a_2, \ldots, a_n\}$. Le groupe or composé de l'ensemble d'attributs $\{a_1, a_2, \ldots, a_n\}$ et de leur parent commun a_0 est un groupe xor si et seulement si $\nexists (C_i, C_j) | C_i, C_j \in \{C_1, C_2, \ldots, C_n\}$ et $X_i \cap X_j \neq \varnothing$.*

Autrement dit, ces extensions forment une partition de l'extension X_0 de C_0 [RPK11].

Sur la taille des groupes

Nous définissons la taille d'un groupe par le nombre de concepts introducteurs d'attributs pris en compte lors de l'extraction du groupe. Sélectionner moins de concepts ne signifie pas sélectionner moins d'attributs, car plusieurs attributs peuvent être introduits dans un même concept. Cette règle fonctionne lorsque l'on garde plusieurs attributs par concept. Si n concepts introducteurs d'attributs couvrent l'extension de C_0, on peut éventuellement extraire plusieurs groupes d'attributs différents de ces mêmes concepts, en sélectionnant à chaque fois un attribut par concept et en réalisant des combinaisons différentes. Pour éviter la redondance, nous considérons alors chaque ensemble d'attributs cooccurrents

comme étant un seul élément. En d'autres termes, nous considérons l'ensemble quotient de la relation d'équivalence "est cooccurrent avec".

Exemple. Nous avons vu précédemment que l'attribut `Data Model` introduit dans le concept 9 est un parent de groupe (car les propriétés 1 et 2 des groupes sont respectées). Si l'on considère le groupe *or* formé par l'ensemble des attributs candidats, alors ce groupe est de taille 5 (car formé des concepts 2, 5, 6, 7 et 8). Il y a cependant 6 caractéristiques dans ce groupe, mais deux d'entre elles sont cooccurrentes : `OS:Mac` et `OS:Linux`. De ce fait, il est possible de garder les deux caractéristiques dans le groupe, ou bien seulement une des deux : nous pouvons alors composer trois groupes différents. Nous choisissons de condenser ces groupes possibles en un seul groupe composé d'une cooccurrence.

Concernant les groupes *or* (non purs), les groupes de taille minimale sont inclus dans le voisinage inférieur du concept introduisant l'attribut parent dans l'AC-poset. Ce sont ses sous-concepts les plus hauts dans le treillis et donc ceux qui possèdent le plus d'objets dans leur extension. Un concept plus bas dans le treillis ne peut pas couvrir plus, car ses objets seraient hérités par ses super-concepts. Il se peut cependant que des concepts de ce voisinage puissent être enlevés du groupe sans rompre la couverture. Un groupe de taille minimale est donc constitué d'un plus petit ensemble de concepts du voisinage inférieur dans l'AC-poset, tel qu'il n'y a aucun concept qui puisse être enlevé sans rompre la couverture. Il peut y avoir plusieurs groupes de taille minimale.

Exemple. Dans la Figure 3.2, les concepts 6, 7 et 8 forment un groupe *or*, mais celui-ci n'est pas de taille minimale. En effet, si l'on enlève le concept 6, le groupe formé par les concepts 7 et 8 couvre entièrement l'extension du concept 9. Les concepts 7 et 8 forment donc un groupe *or* de taille minimale ayant un attribut introduit dans le concept 9 pour parent. Les concepts 5 et 6 forment aussi un groupe *or* minimal sous le concept 9.

Il n'existe qu'un seul groupe de taille maximale : il est composé de tous les sous-concepts introducteurs d'attributs de C_0.

Exemple. Le groupe *or* de taille maximale du concept 9 est composé des attributs introduits dans ses 5 sous-concepts introducteurs d'attributs : les concepts 2, 5, 6, 7 et 8.

Dans un groupe *xor*, aucun concept ne peut être enlevé sans rompre la couverture et aucun concept ne peut être ajouté sans rendre la couverture non exacte ; chaque concept couvre un certain nombre d'objets, qui ne sont pas couverts par les autres concepts sélectionnés. Donc, si l'on ajoute un concept au groupe, ce concept est nécessairement comparable à l'un de ceux du groupe actuel.

En effet, imaginons que l'on a une couverture exacte et qu'il existe un autre concept incomparable, introduisant un attribut. S'il possède un objet dans son intension, l'objet est forcément hérité par C_0 qui est un super-concept. Or tous les objets sont couverts. Le concept ne peut donc pas avoir d'extension vide. Or, si c'est le cas, il correspond au *bottom-concept*, qui est sous-concept de tous les concepts. Cela a deux conséquences. Premièrement, il ne peut donc pas y avoir de concepts incomparables à l'ensemble sélectionné parmi les sous-concepts de C_0. Dans le graphe des mutex restreint aux sous-concepts de C_0, un groupe *xor* correspond donc à une clique maximale [SLB+11]. Deuxièmement, ajouter un concept au groupe *xor* revient à ajouter un concept comparable et donc possédant dans son extension au moins un objet déjà couvert. De ce fait, il ne peut y avoir de groupes *xor* inclus dans un autre.

Les groupes *xor* minimaux sont donc ceux dont les concepts sont les plus en haut du treillis. Ils correspondent aux plus petites des cliques maximales du graphe des mutex. Les groupes maximaux sont ceux dont les concepts sont les plus bas dans le treillis et correspondent aux plus grandes des cliques maximales du graphe de mutex. Il n'y a pas de groupes *xor* dans le treillis de la Figure 3.2.

Notons qu'il est possible de borner la taille des groupes *xor* avant de les calculer. Dans l'AC-poset, il faut pour cela analyser le nombre de concepts minimaux, i.e. les concepts n'ayant pas de sous-concepts. En effet, nous avons vu que si deux concepts introducteurs d'attributs ont leurs extensions disjointes, alors ils n'ont pas de sous-concepts communs dans l'AC-poset (si l'on ne prend pas en compte les attributs qui ne sont possédés par aucun objet). Ces concepts sont donc forcément des super-concepts de deux concepts minimaux différents dans l'AC-poset. Soit q le nombre de concepts minimaux dans un AC-poset, on peut donc déduire qu'il n'y aura pas de groupes *xor* possédant un nombre d'éléments plus grand que q. Si c'était le cas, alors il existerait au moins deux concepts introducteurs d'attributs ayant le même concept minimal de l'AC-poset pour minorant. Dans un treillis de concepts, le nombre q peut être trouvé en calculant le nombre de concepts dans le voisinage supérieur du *bottom-concept*. Par conséquent, si un groupe *or* est composé de plus de q éléments, il n'est pas nécessaire de tester s'il est un groupe *xor*.

Exemple. D'après le treillis de la Figure 3.2 et l'AC-poset de la Figure 3.4, on peut déduire que s'il y avait des groupes *xor* dans notre exemple, ils ne pourraient pas être de taille supérieure à 2.

Algorithmes

L'algorithme 4 présente le test pour savoir si un concept introducteur d'attribut est un parent de groupe.

Algorithme 4 : IsGroupParent

Data : C : the concept introducing the potential parent attribute, AC_K : set of attribute-concepts of a formal context K, $S_r = (C_i, C_j)$, $C_i, C_j \in C_K$ and $C_i <_s C_j$: the cover relationship between the concepts of AC_K, i.e. the transitive reduction of the partial order between concepts of AC_K

Result : $isParent$: a boolean

1 $ext \leftarrow \varnothing$
2 **foreach** $(C_i, C) \in S_r$ **do**
3 $\quad ext \leftarrow ext \cup Extent(C_i)$
4 $isParent \leftarrow (Extent(C) = ext)$

En entrée, cet algorithme prend un concept introducteur d'attributs $C \in AC_K$ et l'AC-poset, représenté par a) l'ensemble des concepts introducteurs d'attributs AC_K et b) l'ordre partiel entre ces concepts défini par des couples $(C_i, C_j), C_i, C_j \in AC_K$ et $C_i <_s C_j$. On ne considère que les couples (C_i, C_j) représentent la relation de couverture entre ces concepts, c'est-à-dire entre un concept et ses super-concepts directs uniquement, ce qui correspond à la réduction transitive de l'AC-poset. L'algorithme renvoie un booléen déterminant si le concept C introduit un attribut parent de groupe.

Pour cela, on parcourt tous les couples concept/super-concept directs (ligne 2) : si C correspond au super-concept C_j, alors C_i est un sous-concept direct de C. Dans ce cas-là, on conserve l'extension du sous-concept dans l'ensemble *ext* (ligne 3) qui, à la ligne 4, représentera l'union de tous les objets présents dans les extensions des sous-concepts directs de C. Si *ext* est égal à l'extension de C, alors C introduit un attribut parent, sinon, il ne peut être le parent d'un groupe (ligne 4).

Le nombre de couples de concepts représentant l'ordre partiel est supérieurement borné par $\frac{|A|(|A|-1)}{2}$, mais décroît avec la profondeur du concept à tester dans l'AC-poset. La complexité dans le pire des cas est de $\mathcal{O}(|A|^2)$.

L'algorithme 5 permet d'extraire tous les groupes *or* (non purs) et *xor* depuis l'AC-poset. Cette extraction peut aussi se réaliser sur les treillis de concepts et l'AOC-poset. Toutefois, nous ne présentons que la version utilisant l'AC-poset, moins complexe et plus compacte à définir.

Algorithme 5 : ExtractGroups

Data : AC_K : set of attribute-concepts of a formal context K, S_{AC_K} : the cover relationship between the concepts of AC_K
Result : xor, or

```
1  T_ACK ← TransitiveClosure(S_ACK)
2  q ← number of minimal concepts in AC_K
3  foreach C ∈ AC_K do
4      if IsGroupParent(C, AC_K, S_ACK) then
5          foreach (C_i, C) ∈ T_ACK) do
6              LC ← LC ∪ {C_i}
7          foreach R ∈ P(LC) do
8              cover ← ⋃_{C_l ∈ R} Extent(C_l)
9              if ext = Extent(C) then
10                 isXor ← true
11                 if |R| ≤ q then
12                     foreach m ∈ {1, 2, ..., |R| − 1} do
13                         foreach n ∈ {m, m + 1, ..., |R|} do
14                             if Extent(C_m) ∩ Extent(C_n) ≠ ∅ then
15                                 isXor ← false
16                 else
17                     isXor ← false
18                 if isXor then
19                     xor ← xor ∪ (C, R)
20                 else
21                     or ← or ∪ (C, R)
```

En entrée, cet algorithme reçoit l'AC-poset, c'est-à-dire l'ensemble des concepts introducteurs d'attributs AC_K et l'ordre partiel $S = (C_i, C_j), C_i, C_j \in AC_K, C_i \leq_s C_j$ entre ces concepts. Il construit deux ensembles, dénommés *or* et *xor* contenant les groupes extraits de l'AC-poset.

Dans un premier temps (ligne 1), il est nécessaire de calculer la fermeture transitive de l'ordre partiel entre les concepts : il nous sera utile plus tard pour retrouver l'ensemble des sous-concepts d'un concept donné. Nous calculons aussi le nombre de concepts minimaux q de l'AC-poset afin de connaître la borne maximale de la taille des groupes *xor* potentiels (ligne 2). Puis, pour chaque concept C de l'AC-poset, nous testons s'il introduit des attributs représentant des parents de groupe (lignes 3-4). S'il représente des parents de groupes, l'ensemble de ses sous-concepts est collecté dans l'ensemble LC : pour cela, chaque couple concept/super-concept (C_i, C) est parcouru (lignes 5-6). Chaque ensemble R correspondant à une partie de l'ensemble de concepts LC est ensuite étudié (ligne 7). La variable *cover* stocke l'union des extensions des concepts de R (ligne 8). Si *cover* représente l'extension du concept C (ligne 9), alors R est un groupe *or*. Nous vérifions ensuite si ce groupe est un groupe *xor*. D'abord, nous testons si la taille du groupe (i.e. $|R|$) est inférieure ou égale à q (ligne 11). Si elle ne l'est pas, alors le groupe ne peut pas être de type *xor* (ligne 17). Sinon, nous testons chaque paire de concepts de R : si toutes les paires de concepts ont l'intersection de leurs extensions vide (lignes 13-15), alors R est un groupe

xor.

L'ensemble des sous-concepts LC est supérieurement borné par $|A|-1$. L'ensemble des parties de LC a donc une cardinalité de $2^{|A|-1}$. L'ensemble des paires d'éléments de R est borné supérieurement par $\frac{|A|(|A|-1)}{2}$. La complexité dans le pire des cas est donc de l'ordre de $2^{|A|} \times |A|^2$.

3.4.5 Comparaison des méthodes d'extraction de la variabilité

Dans cette section, nous présentons les différents travaux qui s'intéressent à l'extraction des informations représentant la variabilité sous la forme de relations logiques. Dans un premier temps, nous discutons des travaux utilisant l'AFC et les structures conceptuelles comme support à l'extraction de ces relations. Puis, nous étudions les travaux se basant sur des formules propositionnelles et utilisant des solveurs SAT. Nous comparons les méthodes utilisées, leurs complexités et nous déterminons si elles sont valides et/ou complètes.

Extraction basée sur l'AFC La Table 3.2 résume les informations relatives aux 3 articles présentés dans cette section, par rapport à la méthode exposée à la section précédente. La validité de la méthode d'extraction d'un type de relation est définie en fonction de la définition de la dite relation dans les FMs. M signifie méthode, P prétraitement et C complexité.

Shatnawi *et al.* [SSS17] proposent une méthode pour extraire les 5 types de relations les plus communs, qui correspondent à ceux étudiés précédemment. Leurs travaux se situent dans le contexte de l'extraction d'information de variabilité au niveau de l'architecture de variantes logicielles. Pour cela, ils utilisent l'AFC et se basent sur la description des éléments composant l'architecture d'une FPL représentée sous forme de contexte formel. Leur extraction des implications binaires et des cooccurrences est identique à la nôtre. Pour détecter les mutex, ils calculent l'ensemble des chaînes du treillis de concepts (i.e. les ensembles de concepts comparables) et cherchent les paires d'éléments introduits dans des concepts qui n'apparaissent jamais ensemble dans une chaîne. Cette méthode d'extraction est complète. Ils considèrent de plus que les groupes *xor* sont une généralisation des mutex, pouvant avoir plus de deux éléments : ils cherchent donc les ensembles d'éléments mutuellement exclusifs deux à deux. Dans la méthode proposée par les auteurs, les parents des groupes *xor* ne sont pas identifiés. Leur définition ne correspond donc pas à la définition des FMs. Enfin, les groupes *or* sont extraits grâce à une heuristique. Dans un premier temps, ils éliminent toutes les paires d'éléments liés par une des relations précédentes : les paires restantes sont notées comme étant des groupes *or*. Ensuite, ils considèrent les paires ayant des éléments communs et modifient les groupes en fonction des relations existantes entre les éléments non communs. Si ces éléments sont liés par une implication binaire, ils fusionnent les deux groupes et gardent l'élément le plus général (i.e. la conclusion de l'implication binaire). Si ces deux éléments sont mutuellement exclusifs, les deux groupes sont retirés de la liste des groupes *or*. Enfin, si ces deux éléments sont dans un autre groupe *or*, il est simplement fusionné avec cet autre groupe. Leur méthode n'extrait pas tous les groupes *or* possibles et ne définit pas de parent : là aussi, leur définition du groupe *or* est différente de celle des FMs. Ils proposent par la suite, à partir de ces relations, de construire un arbre de caractéristiques dans lequel les groupes se verront assigner un parent. La complexité n'est pas discutée en détails, mais est identifiée comme polynomiale. Contrairement à leurs travaux, notre détection des groupes n'est pas polynomiale, car nous avons des définitions différentes de ce que sont les groupes. Notre extraction de groupes est valide et complète, alors que celle présentée dans [SSS17] n'est pas complète et n'est potentiellement pas valide vis-à-vis des groupes tels que définis par les FMs (i.e. leurs

TABLE 3.2 – Extraction des informations sur la variabilité depuis des contextes formels : méthode présentée à la section précédente, Shatnawi *et al.* [SSS17], Al-Msiedeen *et al.* [AMHS$^+$14] et Ryssel et al [RPK11]. M signifie méthode, P prétraitement et C complexité.

		Implications binaires	**Cooccurrences**	**Mutex**	**or (non purs)**	**xor**
Section 3.4	M	Ordre partiel entre les concepts dans l'AC-poset (*complet & valide*)	Attributs introduits dans les mêmes concepts dans l'AC-poset (*complet & valide*)	Paires de concepts incomparables dans l'AC-poset, dont l'intersection des extensions est non vide (*complet & valide*)	Sous-ensembles des sous-concepts incomparables d'un concept parent couvrant son extension (*complet & valide*)	Groupes or dont l'intersection des extensions de chaque paire de concepts est vide (*complet & valide*)
	P	$\mathcal{O}(\|A\|)$ pour la construction de l'AC-poset				
	C	$\mathcal{O}(\|A\|^2)$	$\mathcal{O}(\|A\|)$	$\mathcal{O}(2\|O\|\|A\|^2)$	$\mathcal{O}(2^{\|A\|})$	
[SSS17]	M	Ordre partiel entre les concepts du treillis (*complet & valide*)	Attributs introduits dans les mêmes concepts du treillis (*complet & valide*)	Paires d'attributs n'apparaissant jamais dans une même chaîne du treillis de concepts (*complet & valide*)	Heuristique (*non complet & non valide*)	Groupes d'attributs mutex deux à deux (*non complet & non valide*)
	P	$\mathcal{O}(V+E)$ avec $V \leq 2^{(min(\|O\|,\|A\|)}$ et $E \leq V^2$)				
	C	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\|)$	$\mathcal{O}(\|V\|^3)$	/	/
[AMHS$^+$14]	M	Ordre partiel entre les concepts dans l'AC-poset : extraction de la réduction transitive (*complet & valide*)	Attributs introduits dans les mêmes concepts de l'AC-poset (*complet & valide*)	Ensemble des super-concepts des concepts minimaux de l'AOC-poset (*non complet & valide*)	Faux groupe or, depuis l'AOC-poset (*non complet & non valide*)	Faux groupe xor, depuis l'AOC-poset (*non complet & non valide*)
	P	$\mathcal{O}(\|A\|)$ pour la construction de l'AC-poset, $\mathcal{O}(\|O\|+\|A\|)$ pour l'AOC-poset				
	C	$\mathcal{O}(\|A\|^2)$	$\mathcal{O}(\|A\|)$	$\mathcal{O}(\|A\|+(4\|O\|\|A\|)+\|A\|^2)$	/	/
[RPK11]	M	Ordre partiel entre les concepts de l'AC-poset (*complet & valide*)	/	Ordre partiel entre les concepts de l'AC-poset possédant les attributs et les négations des attributs (*complet & valide*)	Couverture de l'extension d'un concept dans l'AC-poset (*complet & valide*)	Couverture exacte de l'extension d'un concept dans l'AC-poset (*complet & valide*)
	P	$\mathcal{O}(\|A\|)$ pour la construction de l'AC-poset, $\mathcal{O}(2\|A\|)$ pour l'AC-poset avec les négations				
	C	$\mathcal{O}(\|A\|^2)$	/	$\mathcal{O}(2\|A\|^2)$	$\mathcal{O}(2^{\|A\|})$	

auteurs extraient des groupes qui ne correspondent pas à cette définition). Nos méthodes d'extraction des implications binaires, des cooccurrences et des mutex sont plus efficaces car nous nous basons sur l'AC-poset et non pas sur le treillis de concepts.

Al-Msiedeen *et al.* [AMHS$^+$14] extraient ces 5 types de relations depuis l'AC-poset ou l'AOC-poset dans le cadre de travaux sur la synthèse automatique de FMs. Leurs méthodes d'extraction des implications binaires et des cooccurrences correspondent à celles que nous avons présentées dans la section précédente, mais ils ne conservent que la réduction transitive des implications binaires. Ils proposent ensuite une méthode pour extraire un seul groupe *xor*, se basant sur l'identification des concepts minimaux dans l'AOC-poset.

Tout d'abord, chaque caractéristique identifiée comme étant dans une cooccurrence n'est pas prise en compte dans la recherche du groupe *xor*. Ensuite, pour chaque concept minimal, l'ensemble de ses super-concepts est récupéré. De chaque ensemble de super-concepts sont éliminés les concepts qui sont présents dans l'ensemble de super-concepts d'un autre concept minimal. Ainsi ne sont gardés que les super-concepts spécifiques à chaque concept minimal et dont l'intersection des extensions avec des super-concepts d'autres concepts minimaux sera toujours vide. Pour chaque concept minimal, les auteurs conservent le concept introducteur d'attribut le plus général de l'ensemble de ses super-concepts spécifiques. Il correspond à l'ensemble de caractéristiques mutuellement exclusives les plus générales de la structure conceptuelle. Le parent du groupe n'est pas déterminé et peut potentiellement ne pas exister (i.e. on ne peut pas sélectionner plusieurs caractéristiques du "groupe" ainsi extrait, mais rien n'oblige la sélection d'au moins une de ces caractéristiques). C'est donc une définition de groupe *xor* différente de celle que nous étudions. Ensuite, les auteurs proposent une heuristique pour extraire un groupe *or* : ils éliminent les caractéristiques introduites dans le *top-concept* (i.e. présentes dans toutes les configurations), celles impliquées dans les cooccurrences et le groupe *xor*. Les caractéristiques restantes forment un unique groupe *or*. Ainsi, les auteurs cherchent à grouper les caractéristiques qui sont présentes dans certaines variantes, mais pas toutes. C'est une définition des groupes *or* différente de celle étudiée précédemment, car 1) il n'y a pas de recherche de parent de groupe et 2) il manque la contrainte exprimant qu'au moins une des caractéristiques doit être présente dans chaque variante. Enfin, ils extraient les mutex en se basant sur la méthode qu'ils ont définie pour l'extraction du groupe *xor*, mais cette fois ils ne gardent pas uniquement les concepts les plus généraux. Leur méthode d'extraction des mutex n'est pas complète car ils écartent certains attributs, mais elle est valide car les mutex extraits sont corrects. En comparaison, notre méthode d'extraction des mutex est complète et nous extrayons les groupes en suivant la définition des FMs.

Ryssel *et al.* [RPK11] travaillent sur la synthèse de FMs depuis un contexte formel. Ils extraient les implications binaires, les exclusions mutuelles et les deux types de groupes. Ils ne traitent pas les cooccurrences, mais proposent d'extraire des implications non binaires. Ils construisent l'AC-poset pour extraire l'ensemble des implications binaires (i.e. sa fermeture transitive) de la même manière que présenté précédemment. Ils proposent ensuite d'extraire les groupes *or* et *xor*, en se basant sur les définitions étudiées, contrairement aux travaux vus précédemment. Ils résument la problématique de trouver les groupes *or* à celle de trouver des concepts introducteurs d'attributs (groupe) dont les extensions forment une couverture de l'extension d'un autre concept (parent du groupe). La résolution de ce problème est exponentielle en la taille de l'entrée (ici le contexte formel). Similairement, ils définissent l'extraction des groupes *xor* cette fois-ci par la découverte des couvertures exactes. Trouver une couverture exacte est un cas spécial de couverture, qui est aussi un problème exponentiel en la taille de l'entrée. Les auteurs notent cependant que, puisque les groupes *xor* sont un cas particulier de groupes *or*, on peut restreindre la recherche des groupes *xor* aux groupes *or* détectés. Nous extrayons les deux types de groupes en même temps, de la même façon que Ryssel *et al.*, à la différence près que nous réduisons l'espace de recherche des concepts à prendre en compte dans les groupes aux sous-concepts du concept introduisant l'attribut parent. Leur méthode d'extraction des groupes est valide et complète. Pour calculer les mutex, ils introduisent dans le contexte formel de départ la négation de chaque attribut et associent chaque objet à l'ensemble des attributs qu'il possède et à la négation des attributs qu'il ne possède pas. Ils extraient ensuite l'ensemble des implications binaires dans l'AC-poset associé au contexte augmenté par les négations d'attributs. Les implications binaires contenant une négation d'attribut représentent les mutex. Cette méthode permet d'extraire tous les mutex qui sont vrais pour le contexte formel de départ,

sans négations d'attributs. Cette méthode réduit la complexité de l'extraction des mutex par rapport à la méthode que nous avons proposée. Cependant, le nombre d'attributs est doublé, ce qui complexifie la structure conceptuelle finale (possédant plus d'attributs et donc potentiellement plus de concepts), qui doit de plus être recalculée.

La Figure 3.6 représente deux treillis de concepts présentant 5 objets. Celui de gauche décrit ces objets en fonction de 3 attributs. Celui de droite décrit les mêmes objets avec les mêmes attributs, sauf qu'il possède 3 attributs supplémentaires représentant la négation des trois attributs de départ. Par exemple, l'objet *O5* du treillis de gauche est décrit par l'attribut `Att1`. Dans le treillis de droite, ce même objet est décrit par l'attribut `Att1`, mais aussi par les deux attributs qu'il n'a pas, à travers `not Att2` et `not Att3`. Dans cet exemple, l'ajout de la négation des attributs augmente la taille du treillis de concepts de 8 concepts à 15 concepts.

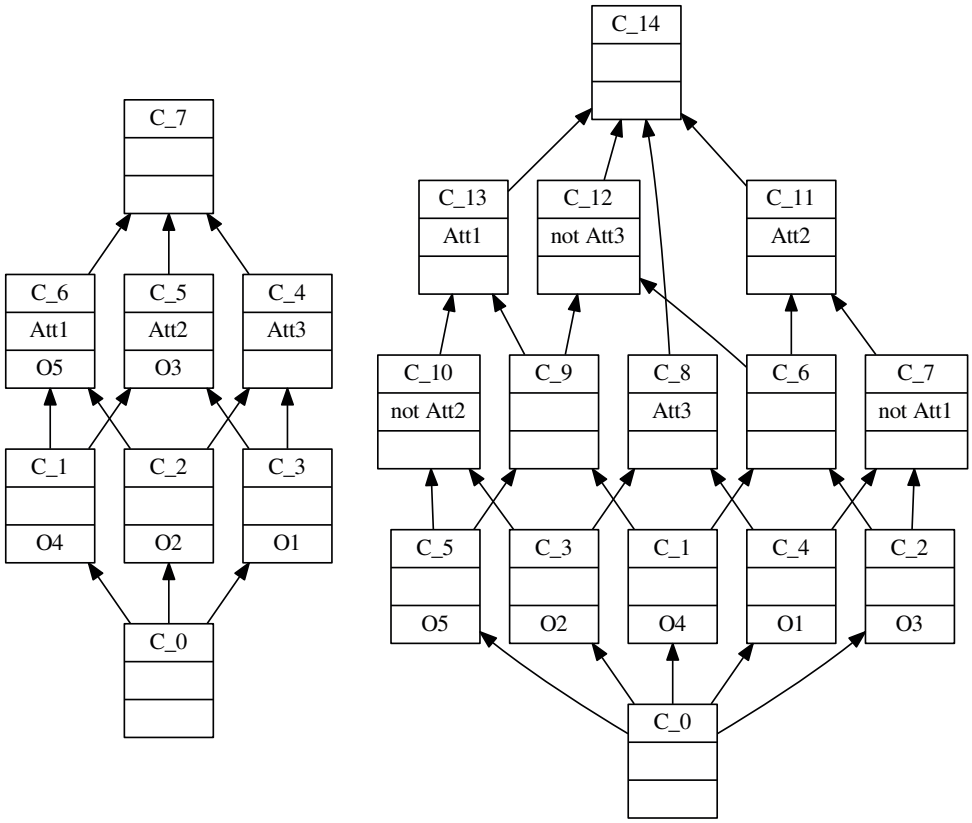


FIGURE 3.6 – (gauche) Treillis de concepts sans négation ; (droite) Treillis de concepts avec négations

Extraction basée sur les formules propositionnelles La Table 3.3 résume les informations relatives aux 3 articles présentés ci-après. *M* signifie méthode et *C* complexité.

Czarnecki et Wasowski [CW07] étudient les FMs du point de vue de leur sémantique logique. Le point de départ de leurs travaux est une formule propositionnelle φ, dont les variables représentent les caractéristiques d'un ensemble de produits logiciels. Ils montrent comment extraire un hypergraphe d'implications à partir de la forme normale conjonctive (CNF) de φ. En effet, chaque clause non vide de la CNF peut être convertie en une implication (pas nécessairement binaire), pouvant être représentée dans un hypergraphe. En ne gardant que les implications binaires de l'hypergraphe d'implications, ils conservent un graphe d'implications binaires, représentant toutes les implications binaires de la formule

TABLE 3.3 – Extraction des informations sur la variabilité depuis des formules propositionnelles : méthodes de Czarnecki et Wasowski [CW07] et de She *et al.* [SLB+11, SRA+14]. M signifie méthode, P prétraitement et C complexité.

		Implications binaires	**Cooccurrences**	**Mutex**	**or (non purs)**	**xor**
[CW07]	M	Graphe d'implications binaires d'une formule propositionnelle (*complet & valide*)	Cliques maximales dans le graphe d'implications binaires (*complet & valide*)	/	Groupes minimaux par calcul des implicants premiers (*non complet & valide*)	Groupes or avec des variables deux à deux mutuellement exclusives (*complet & valide*)
	C	/	$O(V+E)$ avec $V=\|A\|$, $E \leq \|A\|^2$	/	/	/
[SLB+11]	M	Graphe d'implications binaires d'une formule propositionnelle φ (*complet & valide*)	/	Arête (u,v) dans le graphe des mutex telle que $\varphi \models (u \rightarrow \neg v)$ (*complet & valide*)	/	Cliques maximales dans le graphe des mutex et vérification de la présence d'au moins une caractéristique du groupe avec le parent (*complet & valide*)
	C	/	/	/	/	/
[SRA+14]	M	Graphe d'implications binaires d'une formule propositionnelle φ (*complet & valide*)	Cliques maximales dans le graphe d'implications binaires (*complet & valide*)	Arête (u,v) dans le graphe des mutex tel que $\varphi \models (u \rightarrow \neg v)$ (*complet & valide*)	Groupes minimaux par calcul des implicants premiers (*non complet & valide*)	Groupes or formant une clique dans le graphe des mutex (*complet & valide*)
	C	$O(\|\varphi\|\|A\|^2)$	/	$O(\|\varphi\|\|A\|^2)$	/	/

propositionnelle de départ. La complexité n'est pas discutée. Ils identifient les cooccurrences dans le graphe d'implications binaires en cherchant les cliques maximales. Cela peut se faire en $O(V+E)$, avec $V=|A|$ et $E \leq \frac{|A|(|A|-1)}{2}$. Les auteurs définissent ensuite le problème de détection des groupes *or* par le calcul des implicants premiers d'une formule propositionnelle. Soit f une variable dans φ, les implicants premiers de $\varphi \rightarrow f$ qui sont consistants avec φ représentent les groupes *or* minimaux ayant f comme caractéristique mère. Le calcul des implicants premiers ne donne que les groupes *or* minimaux, cette méthode d'extraction n'est donc pas complète. Ils définissent les groupes *xor* comme des groupes *or* dont les caractéristiques sont mutuellement exclusives. Puisque, par définition, il n'existe pas de groupes *xor* inclus dans d'autres groupes *xor*, les groupes *xor* sont tous minimaux (et maximaux), cette méthode est donc complète. Pour l'extraction des deux types de groupes, les complexités ne sont pas discutées. Enfin, les exclusions mutuelles ne sont pas traitées, mis à part pour le calcul des groupes *xor*.

Dans [SLB+11], les auteurs étudient aussi la synthèse de FMs depuis une formule propositionnelle φ. De la même manière que Czarnecki et Wasowski [CW07], un graphe d'implications binaires est calculé depuis la formule propositionnelle. L'ensemble des mutex est représenté sous la forme d'un graphe de mutex, dont les noeuds représentent les caractéristiques et où une arête entre deux nœuds indique qu'ils sont mutuellement exclusifs. Une exclusion mutuelle entre deux variables u et v de la formule est déterminée en vérifiant que $\varphi \models (v_1 \rightarrow \neg v_2)$: cette méthode est complète car elle permet de déterminer tous les

mutex. Ce graphe de mutex est aussi utilisé pour détecter les groupes *xor*. Ils extraient les groupes *xor* ayant la caractéristique f comme parent en détectant les cliques maximales parmi les caractéristiques filles de f dans le graphe des mutex. Ensuite, ils vérifient que, lorsque f est présente, au moins une des caractéristiques du groupe est aussi toujours présente. Cette méthode extrait tous les mutex. Les groupes *or* et les cooccurrences ne sont pas étudiés dans cet article. Aucune complexité n'est discutée.

Dans [SRA+14], les auteurs complètent les méthodes d'extraction présentées par Czarnecki et Wasowski [CW07] avec les méthodes utilisant le graphe des mutex de [SLB+11]. À partir d'une formule propositionnelle φ, ils construisent le graphe d'implications binaires en vérifiant si pour chaque paire (u, v), $\varphi \models (u \rightarrow v)$. Ce graphe peut être construit en $O(|\varphi||A|^2)$. Là aussi, les cooccurrences sont extraites en repérant les cliques maximales du graphe d'implications binaires. La complexité n'est pas discutée. Le graphe des mutex est ensuite construit de telle sorte que pour chaque paire (u, v) de variables représentant les caractéristiques, $\varphi \models (u \rightarrow \neg v)$. Le graphe des mutex peut être construit en $O(|\varphi||A|^2)$. Les groupes *or* sont extraits en calculant les implicants premiers tel que présenté dans [CW07]. Là encore, cette méthode n'extrait pas tous les groupes *or* possibles. Enfin, les groupes *xor* sont détectés parmi les groupes *or* dont l'ensemble des variables du groupe correspond à une clique dans le graphe des mutex. L'extraction des groupes *xor* est complète.

En résumé (3.4) :

- Nous avons montré comment extraire les **implications binaires**, **cooccurrences**, **mutex** et **groupes** en utilisant l'**analyse formelle de concepts**.
- Nous avons montré que l'**AC-poset**, un sous-ordre du treillis de concepts ne conservant que les concepts introducteurs d'attributs, est suffisant pour extraire les 5 relations représentant la variabilité des FMs.
- Les **méthodes d'extraction** proposées dans ce chapitre sont **complètes et valides**.
- Nous avons fait une **comparaison** des méthodes d'extraction d'information de variabilité sous forme de relations logiques présentes dans la littérature avec les méthodes définies et ce en terme de **validité**, de **complétude** et de **complexité**.

Nous venons de voir comment extraire les 5 types de relations logiques correspondant à la sémantique logique des FMs depuis des structures conceptuelles. Dans ce qui suit, nous nous appuyons sur ces types de relations pour établir des corrélations entre les FMs et les structures de base de l'AFC.

3.5 Corrélations avec les modèles de caractéristiques

Dans cette section, nous étudions les corrélations entre les FMs (présentés dans la Section 2.4) et les structures conceptuelles de l'AFC (présentées dans la Section 3.2). Pour cela, nous étudions les différentes sémantiques de ces deux formalismes (Sections 3.5.1 et 3.5.2) et plus particulièrement leur sémantique logique, pour en réaliser la comparaison (Section 3.5.3). Nous montrons ainsi qu'un treillis de concepts représente une classe d'équivalence de modèles de variabilité.

3.5.1 Sémantique logique des FMs

Nous avons vu en Section 2.4 que les FMs possèdent trois sémantiques :

— une *sémantique de configurations*, représentant l'ensemble des configurations valides délimitées par les contraintes du modèle ;
— une *sémantique ontologique*, représentant le sens ontologique des contraintes exprimées par le modèle ;
— une *sémantique logique*, représentant les contraintes exprimées par le modèle sous forme de relations logiques.

Chaque type de relations ontologiques possède une représentation sous forme de logique des propositions, parmi 5 types de relations logiques. Les colonnes `FMs` et φ des Tables 3.4, 3.5 et 3.6 représentent cette correspondance de manière plus détaillée qu'en Section 2.4. Les FMs comptent 8 types de relations ontologiques et 5 types de relations logiques. La colonne `FMs` représente les 8 types de relations logiques. Les relations ayant une représentation graphique (1,2,4,7 et 8) sont illustrées par leur décoration d'arête correspondante et les contraintes transverses (3, 5 et 6) sont identifiées par leur notation textuelle. La colonne φ représente leur sémantique logique [Bat05, CW07, PSA+12, SRA+14]. La dernière colonne `Treillis des concepts` est discutée plus tard.

La ligne **(1)** de la Table 3.4 représente la relation ontologique de parenté d'un couple de caractéristiques et signifie ici que f_2 est la descendante de f_1 (i.e. f_2 raffine f_1). La notation $f_2 <_{FM} f_1$ indique que f_2 est introduite dans la branche de f_1, mais à un niveau strictement plus bas.

La ligne **(2)** montre une relation optionnelle entre f_1 et sa descendante directe f_2. Ce type de relation n'exprime aucune contrainte lors de la sélection, hormis la relation de parenté qui lui est intrinsèque.

La ligne **(3)** représente simplement une contrainte d'implication transverse de f_2 vers f_1.

Ces trois relations ontologiques ont la même sémantique logique et se traduisent par une implication logique $f_2 \rightarrow f_1$.

La ligne (**4**) (gauche) représente une relation obligatoire entre f_1 et sa descendante directe f_2 qui, en plus de la relation de parenté, exprime l'obligation de sélectionner f_2 avec f_1.

La ligne (**5**) expose le cas particulier des implications transverses circulaires.

Ces deux types de relations ontologiques partagent la même sémantique logique et se traduisent par une équivalence logique $f_1 \leftrightarrow f_2$.

La ligne (**6**) montre une exclusion transverse, qui s'expriment par la formule logique $f_1 \rightarrow \neg f_2$ (ou son équivalent $f_2 \rightarrow \neg f_1$).

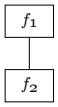
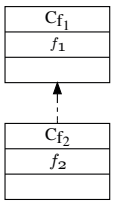
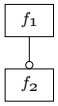
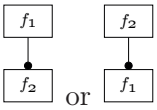
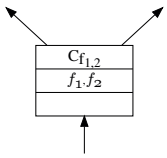
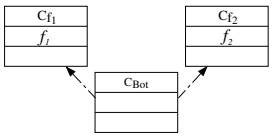
La Table 3.5 présente la correspondance entre les groupes *or* (non purs) et la logique des propositions. $F = \{f_1, \ldots, f_k\}$ est un ensemble de descendantes directes de la caractéristique f_0 qui sont groupées par une relation *or*. Ce type de relation stipule que toutes les configurations valides ayant f_0 ont aussi au moins une des caractéristiques de F et se traduit par la formule logique $f_0 \rightarrow (f_1 \vee \ldots \vee f_k)$.

Enfin, la Table 3.6 présente la correspondance avec les groupes *xor*. Les caractéristiques incluses dans un groupe *xor* ont un comportement similaire à celles faisant partie d'un groupe *or*, excepté qu'elles sont en plus mutuellement exclusives. Cela signifie que, dans toutes les configurations valides ayant f_0, il y a aussi exactement une caractéristique de F. Les groupes *xor* sont représentés ainsi en logique des propositions : $f_0 \rightarrow (f_1 \oplus \ldots \oplus f_k)$.

L'assymétrie de ce mapping donne lieu à la proposition suivante :

Proposition 3.14. *Une relation ontologique d'un FM peut être associée à une seule relation logique, mais une relation logique peut être associée à plusieurs types de relations ontologiques différents.*

TABLE 3.4 – Correspondances entre les relations exprimées par les FMs, les formules de la logique des propositions (ici dénotées par φ) et les treillis des concepts (excepté les groupes de caractéristiques)

	FMs		φ	**Treillis des concepts**	
1		$f_2 <_{FM} f_1$	$f_2 \rightarrow f_1$	$C_{f_2} <_s C_{f_1}$	
2		$optional(f_1, f_2)$			
3	$f_2 \rightarrow f_1$	f_2 *requires* f_1			
4		$mandatory(f_1, f_2)$ or $mandatory(f_2, f_1)$	$f_1 \leftrightarrow f_2$	$C_{f_1} =_s C_{f_2}$	
5	$f_1 \rightarrow f_2$ $f_2 \rightarrow f_1$	f_1 *requires* f_2 f_2 *requires* f_1			
6	$f_1 \rightarrow \neg f_2$	$exclude(f_1, f_2)$	$f_1 \rightarrow \neg f_2$ or $f_2 \rightarrow \neg f_1$	$Ext(C_{f_1} \sqcap C_{f_2}) = \varnothing$	

3.5.2 Sémantique logique des treillis de concepts

Puisque l'ensemble des objets d'un contexte formel (i.e. les configurations valides dans notre cas) se retrouve dans les extensions des concepts des structures de l'AFC, ces dernières présentent une sémantique de configurations à l'instar des FMs. Comme défini en Section 3.4, les structures conceptuelles mettent en évidence des relations logiques entre ces éléments et elles incluent donc une sémantique logique. Notons aussi que l'ensemble des modèles d'une formule propositionnelle peut être représenté sous forme tabulaire par un contexte formel (à la manière d'une table de vérité), où les objets représentent les modèles et les attributs les variables propositionnelles. De ce fait, le treillis des concepts associé à un tel contexte peut être interprété comme un cadre structurel et canonique, car il inclut à la fois la formule propositionnelle (pouvant être extraite de la structure) et ses modèles (au sein de l'extension des concepts).

Proposition 3.15. *Considérons n'importe quelle formule propositionnelle $\mathcal{F}$ sur un ensemble de variables V, ainsi que l'ensemble de ses modèles $\mathcal{M}_\mathcal{F}$. Il existe un unique treillis de concepts $\mathcal{L}_\mathcal{F}$ construit sur le contexte formel $K = (\mathcal{M}_\mathcal{F}, V, R)$ avec $(\forall m \in \mathcal{M}_\mathcal{F}, \forall v \in V, m(v) = true \iff (m, v) \in I)$.*

Réciproquement, considérons un treillis de concepts $\mathcal{L}$ et $K = (O, A, I)$ son contexte formel associé. On peut associer à $\mathcal{L}$ un ensemble de formules propositionnelles équivalentes, construites sur A comme ensemble de variables propositionnelles et telles que l'ensemble de leurs modèles et O soient équipotents. Pour chaque modèle m associé à $o \in O$, on a alors $(\forall a \in A, m(a) = true \iff (o, a) \in I)$.

Cependant, les treillis de concepts ne mettent en évidence que des relations logiques et ne représentent pas explicitement le sens ontologique de ces relations : ils ne possèdent donc pas de sémantique ontologique dans la présente approche.

Les colonnes `Treillis de concepts` et φ des Tables 3.4, 3.5 et 3.6 présentent les

TABLE 3.5 – Mapping : groupes *or* (non purs) (**7**)

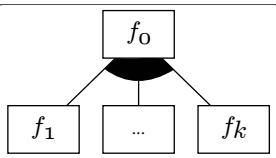
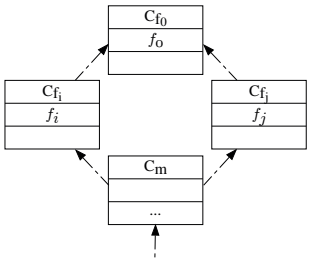
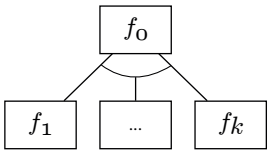
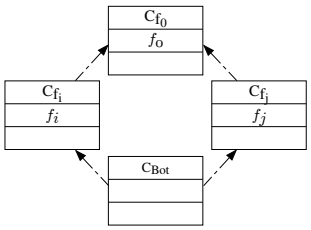
FMs	
	$or(f_0, F) \mid F = \{f_1, f_2, ..., f_k\}$
Formule propositionnelle (φ) : $f_0 \rightarrow (f_1 \vee \ldots \vee f_k)$	
Treillis des concepts	
	$\forall f \in F, C_f \leq_s C_{f_0}.$ $\forall f_i, f_j \in F \mid f_i \neq f_j,\ Ext(C_{f_i} \sqcap C_{f_j}) \neq \varnothing.$ $\bigcup_{f_i \in F} Ext(C_{f_i}) = Ext(C_{f_0}).$ $\forall f_i \in F, (\bigcup_{f_j \in F} Ext(C_{f_j})) \setminus Ext(C_{f_i}) \neq Ext(C_{f_0}).$ $C_{f_0} \notin \mathcal{OC}.$

TABLE 3.6 – Mapping : groupes *xor* (**8**)

FMs	
	$xor(f_0, F) \mid F = \{f_1, f_2, ..., f_k\}$
Formule propositionnelle (φ) : $f_0 \rightarrow (f_1 \oplus \ldots \oplus f_k)$	
Treillis des concepts	
	$\forall f \in F, C_f \leq_s C_{f_0}.$ $\forall f_i, f_j \in F \mid f_i \neq f_j,\ Ext(C_{f_i} \sqcap C_{f_j}) = \varnothing.$ $\bigcup_{f_i \in F} Ext(C_{f_i}) = Ext(C_{f_0}).$ $C_{f_0} \notin \mathcal{OC}$

différentes relations logiques que l'on peut lire dans un treillis et des patterns correspondant à ces relations dans la structure. Cela résume l'analyse théorique présentée en Section 3.4 et nous discutons cette fois les patterns correspondant à ces relations dans les structures conceptuelles.

Lignes (**1,2,3**) : L'implication binaire $f_2 \rightarrow f_1$ est vraie si et seulement si le concept C_{f_2} introduisant la caractéristique f_2 est un sous-concept d'un concept distinct C_{f_1} introduisant f_1 (noté $C_{f_2} <_s C_{f_1}$), i.e. C_{f_2} apparaît en dessous de C_{f_1} dans la structure conceptuelle. Cela est vrai pour le treillis de concepts, l'AOC-poset et l'AC-poset.

Lignes (**4,5**) : L'équivalence $f_1 \leftrightarrow f_2$ est vraie si et seulement si le concept C_{f_1} introduisant la caractéristique f_1 et le concept C_{f_2} introduisant f_2 sont en réalité le même concept (noté $C_{f_1} =_s C_{f_2}$). Les deux caractéristiques sont donc introduites dans le même concept C_{f_1,f_2} et apparaissent toujours ensemble dans une configuration valide ; cela est vrai pour le treillis de concepts, l'AOC-poset et l'AC-poset.

Ligne (**6**) : L'exclusion mutuelle $f_1 \rightarrow \neg f_2$ est vraie si et seulement si l'intersection des extensions des concepts C_{f_1} et C_{f_2}, introduisant respectivement les caractéristiques f_1 et

f_2, est vide. Le plus grand minorant de deux concepts étant le concept dont l'extension est égale à l'intersection de leurs extensions, on peut donc déterminer une exclusion mutuelle en étudiant l'extension du plus grand minorant de deux concepts introducteurs de caractéristiques. Dans un treillis de concepts, le seul concept pouvant avoir une extension vide est le *bottom-concept* (noté C_{Bot} dans la Table 3.4) : s'il est le plus grand minorant de C_{f_1} et C_{f_2} et que son extension est vide, alors f_1 et f_2 sont mutuellement exclusives. Si le *bottom-concept* n'introduit ni caractéristique ni configuration, il n'apparaît donc pas dans l'AOC-poset et l'AC-poset. Dans l'AOC-poset, si C_{f_1} et C_{f_2} n'ont pas de minorant, alors f_1 et f_2 sont mutuellement exclusives. Ce n'est cependant pas vrai pour l'AC-poset : deux concepts peuvent avoir des configurations communes sans pour autant avoir de minorant ; dans ce cas, il faut donc vérifier l'intersection des extensions. Si le *bottom-concept* possède une extension vide mais qu'il introduit au moins une caractéristique, il sera présent dans l'AOC-poset et l'AC-poset. Dans ce cas, si le *bottom-concept* est le plus grand minorant de deux introducteurs de caractéristiques, ces dernières sont mutuellement exclusives.

La Table 3.5 présente la représentation de la sémantique logique des groupes *or* (non purs) dans les treillis de concepts. Soit F l'ensemble des sous-caractéristiques de f_0 formant un groupe *or*. Chaque $f_i \in F$ est introduite dans un concept C_{f_i} qui est un sous-concept de C_{f_0}. Chaque caractéristique du groupe pouvant apparaître avec les autres dans des configurations valides, aucune paire de ces caractéristiques n'est donc une exclusion mutuelle : le plus grand minorant de chaque paire possède donc une extension non vide. L'union des extensions des concepts introduisant les caractéristiques de F est égale à l'extension de C_{f_0}, mais il n'existe aucune caractéristique de F que l'on puisse retirer sans invalider cette propriété. Enfin, C_{f_0} n'introduit pas de configuration.

La Table 3.6 présente la représentation de la sémantique logique des groupes *xor* dans les treillis de concepts. Comme pour les groupes *or*, chaque $f_i \in F$ est introduite dans un concept C_{f_i} qui est un sous-concept de C_{f_0}. Là aussi, l'union des extensions des concepts introducteurs des caractéristiques de F est égale à l'extension de C_{f_0}, mais il n'existe aucune caractéristique de F que l'on puisse retirer sans invalider cette propriété. Mais cette fois-ci, le plus grand minorant de chaque paire de concepts introducteurs correspond au *bottom-concept* dont l'extension est vide.

3.5.3 Analyse des corrélations entre les deux formalismes

Les deux sous-sections précédentes nous permettent d'établir une application entre les relations ontologiques exprimées par un FM et les relations logiques mises en évidence par les structures conceptuelles. On formule naturellement la proposition suivante :

Proposition 3.16. *Tous les types de relations logiques représentant la sémantique logique d'un FM peuvent être lues et extraites dans des treillis de concepts, des AOC-posets et des AC-posets.*

Soit un FM $\mathcal{FM}$ et $K_{\mathcal{FM}}$ le contexte formel représentant l'ensemble exact des configurations valides de $\mathcal{FM}$. Les structures conceptuelles associées à $K_{\mathcal{FM}}$ représentent donc la même sémantique de configurations que $\mathcal{FM}$. De plus, nous avons vu que l'AFC construit des structures conceptuelles canoniques, i.e. il n'existe qu'un seul treillis de concepts (resp. AOC-poset et AC-poset) associé à un contexte formel donné. On peut donc formuler la propriété suivante :

Proposition 3.17. *Soit un FM $\mathcal{FM}$, F son ensemble fini de caractéristiques et v_c son ensemble de configurations valides. Soit le contexte formel $K_{\mathcal{FM}} = (v_c, F, I)$ tel que $\forall c \in v_c, \forall f \in F, f \in c \iff (f, c) \in I$. Il existe toujours un unique treillis de concepts (resp. AOC-poset et AC-poset) associé à $K_{\mathcal{FM}}$ et ayant donc la même sémantique de configuration que $\mathcal{FM}$.*

Les méthodes d'extraction d'information sur la variabilité basées sur l'AFC et qui sont présentées en Section 3.4 ont été prouvées correctes et complètes : toutes les relations logiques (parmi les 5 présentées dans les Tables 3.4, 3.5 et 3.6) qui sont vraies pour l'ensemble de configurations valides considéré sont donc extraites. Soit $CL_{\mathcal{FM}}$ le treillis de concepts associé à $K_{\mathcal{FM}}$. Puisque $CL_{\mathcal{FM}}$ a la même sémantique de configurations que $\mathcal{FM}$, l'intégralité de la sémantique ontologique de $\mathcal{FM}$ peut être extraite de $CL_{\mathcal{FM}}$.

Proposition 3.18. *Soit un FM $\mathcal{FM}$ et soit le treillis de concepts $CL_{\mathcal{FM}}$ ayant la même sémantique de configurations que le modèle. $CL_{\mathcal{FM}}$ est une structure unique contenant l'intégralité de la logique sémantique du FM $\mathcal{FM}$.*

Plus précisément, il est possible de faire correspondre chaque relation ontologique de $\mathcal{FM}$ avec l'un des 5 patterns présentés précédemment dans le treillis de concepts $CL_{\mathcal{FM}}$ et ce en fonction de sa sémantique logique. Cependant, du fait de la Proposition 3.14, plusieurs paires relation logique / relation ontologique sont possibles. Par conséquent, la sémantique logique d'un treillis de concepts peut être associée à plusieurs sémantiques ontologiques différentes, qui correspondent donc à plusieurs FMs différents. Bien que différents au niveau ontologique, ces FMs partagent cependant la même sémantique logique et donc la même sémantique de configurations : on dit qu'ils sont différents, mais équivalents. En effet, nous avons vu dans la Section 2.4 que ces modèles ne sont pas canoniques, i.e. plusieurs modèles différents peuvent représenter le même ensemble de configurations. La construction du treillis reposant sur un ensemble de configurations, l'aspect ontologique n'a donc pas d'influence sur le treillis associé à un modèle. Le treillis de concepts $CL_{\mathcal{FM}}$ inclut donc tous les FMs ayant v_c pour ensemble de configurations valides :

Proposition 3.19. *Soit FM l'ensemble des FMs ayant F comme ensemble fini de caractéristiques et v_c comme ensemble de configurations valides. Soit le contexte formel $K_{\mathcal{FM}} = (v_c, F, I)$ tel que $\forall c \in v_c, \forall f \in F, f \in c \iff (f, c) \in I$. Il existe toujours un unique treillis de concepts (resp. AOC-poset et AC-poset) associé à $K_{\mathcal{FM}}$ et ayant donc la même sémantique de configurations (et donc sémantique logique) que tous les modèles de FM.*

Une structure conceptuelle représente donc une classe d'équivalence de modèles de caractéristiques. Elle structure la variabilité des modèles équivalents sans prendre en compte la dimension ontologique. Cela permet ensuite de dériver des modèles équivalents en associant une sémantique ontologique à la sémantique logique mise en évidence par les structures, comme illustré dans la Figure 3.7

En résumé (3.5) :

- La sémantique logique des FMs s'exprime à travers 5 types de relations logiques différents.
- Ces 5 types de relations peuvent être extraits depuis les treillis de concepts, les AOC-posets ou les AC-posets.
- On peut construire un treillis de concepts (resp. un AOC-poset ou un AC-poset) **unique** depuis un contexte formel représentant les configurations valides d'un FM.
- La structure unique obtenue **inclut toute la sémantique logique (et donc toute la sémantique de configurations)** du FM initial, ainsi que de tous les autres modèles ayant une sémantique logique (et donc de configurations) équivalente.

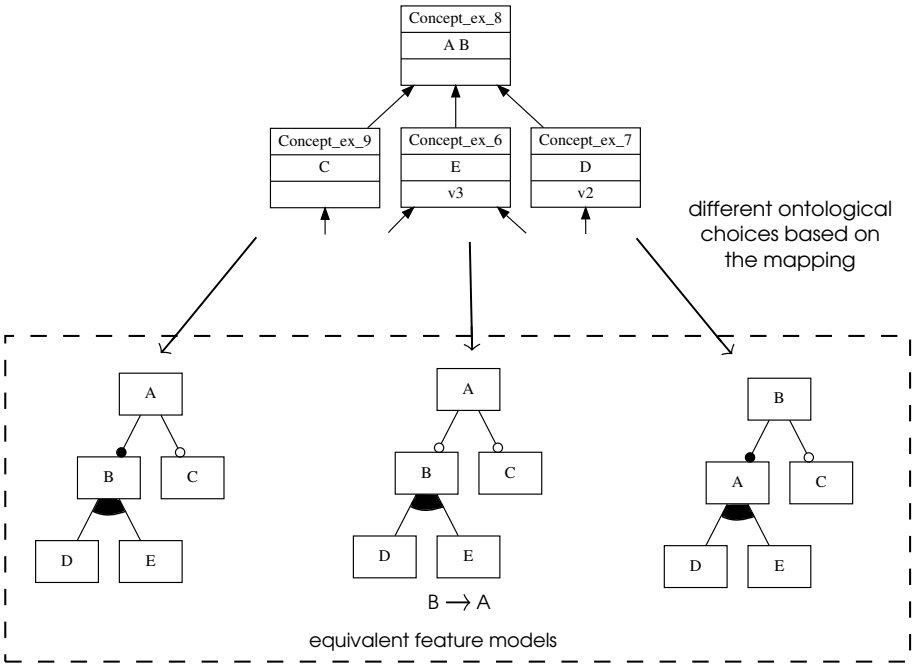


FIGURE 3.7 – Un treillis de concepts représente une classe d'équivalence de FMs

— Ces trois structures conceptuelles représentent donc **des classes d'équivalence de FMs ayant des sémantiques logique et de configurations équivalentes, mais des sémantiques ontologiques différentes**.

3.6 Conclusion

Dans ce chapitre, nous avons étudié les informations relatives à la variabilité que l'on peut lire dans les structures conceptuelles de l'AFC.

Dans un premier temps, nous avons défini formellement l'AFC, son format d'entrée, de sortie, ses mécanismes et ses notions clés. Nous nous sommes concentrés sur 4 structures conceptuelles qui font partie des formats de sortie de l'AFC. Nous avons ensuite analysé ces structures de trois points de vues différents : les concepts (i.e. groupes), la place de ces concepts les uns par rapport aux autres (i.e. hiérarchie entre ces groupes) et plus généralement la place de ces concepts dans la structure (i.e. place des groupes dans la hiérarchie). Pour chacun de ces points de vue, nous avons constaté les différentes informations sur la variabilité qu'ils mettaient intrinsèquement en évidence. Parmi ces informations, certaines caractérisent les interactions entre les caractéristiques et peuvent être représentées sous forme de relations logiques sur l'ensemble des caractéristiques, qui sont vraies pour l'ensemble des configurations étudiées. Nous avons montré comment extraire les 5 types de relations exprimant la sémantique logique des FMs depuis ces structures. Les méthodes d'extraction proposées sont valides et complètes et comparées avec les méthodes présentes dans la littérature. Enfin, à partir de ces observations, nous étudions les corrélations entre les FMs et les structures conceptuelles de l'AFC. Nous montrons qu'il existe une structure unique associée à une sémantique logique, qui inclut donc la variabilité de tous les FMs ayant une sémantique logique (et donc de configurations) équivalente, bien que leurs sémantiques ontologiques puissent être différentes.

En résumé, l'AFC et ses structures conceptuelles forment un cadre qui inclut naturelle-

ment la variabilité des FMs. Ces représentations à la fois intensionnelles et extensionnelles de la variabilité font d'elles un cadre propice à l'étude de la variabilité de variantes existantes. C'est donc un cadre structurel représentant les classes d'équivalence des FMs pouvant représenter la famille de logiciels construite sans effort de réutilisation systématique. De plus, lors de notre étude, nous avons mis en évidence des propriétés qui peuvent se révéler utiles pour gérer la variabiltié, notamment concernant les configurations partielles maximales et les nœuds de décisions représentés par les concepts pour la sélection de configuration. Dans le chapitre suivant, nous étudions la réutilisabilité de ce cadre et de ses propriétés dans des traitements de gestion, d'analyse et de manipulation de la variabilité.

Chapitre 4

L'analyse formelle de concepts pour encadrer la gestion de la variabilité d'une famille de produits logiciels

Préambule

Dans le Chapitre 3, nous avons étudié l'analyse formelle de concepts et ses structures conceptuelles en tant que représentations naturelles de la variabilité. Nous avons identifié les différentes informations que ces structures mettent en évidence et qui permettent de considérer ces structures comme des modèles de variabilité à part entière. Dans ce chapitre, nous souhaitons montrer leur aptitude à servir de support réutilisable à des opérations de traitement de la variabilité et donc à jouer leur rôle de modèles de variabilité. Pour cela, nous proposons une implémentation basée sur l'analyse formelle de concepts de trois opérations connues dans le domaine des lignes de produits logiciels, qui sont la synthèse et la composition de modèles de caractéristiques, ainsi que la sélection de produits.

Sommaire

4.1 Introduction

Comme nous l'avons vu précédemment, les FMs jouent un rôle crucial dans la gestion de la variabilité d'une LPL, car ils sont le support prévalent à nombre d'opérations permettant de manipuler ou d'analyser cette variabilité. Pour rappel, ces opérations incluent (mais ne sont pas limitées à) : la représentation de la variabilité [KCH+90], la recherche d'informations [BSRC10], la dérivation d'architectures logicielles réutilisables et composables [SSS17, XXJ12], les opérations d'analyse et de conception, la maintenabilité de la LPL [ABH+13], son évolution ou encore la configuration et la dérivation de produits logiciels [CHE04, Rab09].

Ces opérations peuvent être d'une grande aide durant la migration (partielle ou totale) vers des approches basées sur la réutilisation systématique et la personnalisation de masse. La migration totale depuis une FPL vers une LPL est un processus long et souvent itératif. Une étude sur les pratiques industrielles montre que les deux stratégies d'adoption de LPLs les plus utilisées sont extractives et réactives [BRN+13]. Durant toute la durée de cette migration, les concepteurs et autres acteurs de la migration vont devoir gérer d'un côté une FPL en potentielle évolution et de l'autre côté une LPL en construction basée sur cette FPL existante. Pouvoir s'appuyer sur diverses opérations de gestion de la variabilité malgré un modèle instable, voire incorrect ou inexistant permet de faciliter et d'accélérer cette migration. La synthèse automatique d'un FM depuis des descriptions de produits a par exemple attiré beaucoup d'attention de la part de la communauté ces dernières années. Notons aussi que les opérations d'analyse et de conception sont des outils facilitant la mise en place de la migration. Dans le cas d'une migration partielle, le but poursuivi n'est pas d'obtenir une LPL à partir d'une FPL, mais de profiter des avantages de ce paradigme en adaptant certaines de ses opérations à la gestion d'un ensemble de logiciels similaires. Par exemple, des catalogues de produits développés sans effort de réutilisation systématique pourraient bénéficier des méthodes de configuration de produits des LPLs, pour aider un utilisateur à choisir un produit répondant à ses exigences.

Dans ce chapitre, nous étudions trois opérations pouvant se révéler utiles lors de la migration partielle ou totale d'une FPL. Nous travaillons avec l'idée que ces opérations doivent pouvoir s'appliquer à des FPLs n'ayant pas de FMs. La première est la synthèse de FMs depuis des descriptions de produits existants (Section 4.2). La deuxième est la composition de modèles, pouvant servir à des fins de conception et/ou d'analyse (Section 4.3). C'est un traitement important dans la gestion des LPLs : nous l'adaptons ici pour qu'il soit utilisable dans le cadre de la migration. La troisième et dernière opération est la sélection de produits (Section 4.4). C'est un processus qui a pour but de guider les décisions d'un utilisateur durant le choix d'une configuration valide de la LPL, qui pourra être dérivée par la suite en une variante logicielle fonctionnelle. Nous l'adaptons pour qu'elle soit applicable à des catalogues de variantes existantes, en proposant une stratégie contournant les limitations des méthodes actuelles. Le point commun de ces trois traitements est qu'ils sont tous mis en place grâce à la représentation de la variabilité fournie naturellement par les structures conceptuelles.

Dans le chapitre précédent, nous avons étudié les qualités de l'AFC pour la représentation de la variabilité d'une famille de logiciels. En particulier, nous avons mis en évidence le fait que les structures de l'AFC incluent de façon inhérente la sémantique logique et la sémantique de configurations des FMs, ce qui fait de l'AFC un **cadre structurel de représentation de la variabilité**. Dans ce chapitre, nous nous concentrons sur la **réutilisation** de ce cadre, en étudiant sa capacité à supporter naturellement plusieurs opérations différentes.

D'autres opérations que les trois présentées ici peuvent bénéficier du support offert par

l'AFC. On trouve dans la littérature des exemples d'application de l'AFC pour gérer la variabilité, comme par exemple extraire l'architecture logicielle d'une LPL [SSS17], améliorer un FM existant [LP07], ou encore localiser les caractéristiques dans du code source [XXJ12, AMSH+13] et en réaliser la traçabilité [SSD13]. Parmi les trois que nous étudions, la synthèse de FMs a déjà été étudiée du point de vue de l'AFC [RPK11, AMHS+14] : nous complétons ici les travaux existants. À notre connaissance, la composition de modèles et la configuration de produits n'ont jamais été implémentées à travers les structures conceptuelles comme nous le proposons.

4.2 Guider la synthèse de FMs

La synthèse de FMs est certainement l'opération la plus étudiée pour la migration vers des LPLs. C'est une tâche ardue qui consiste à définir, à partir de descriptions de variantes de logiciels, un modèle compact et cohérent dont la sémantique de configurations capture le plus fidèlement possible les variantes de départ. Définir ces modèles de variabilité est essentiel dans le processus de migration vers des LPLs, car la plupart des opérations de ce paradigme se basent sur la variabilité et donc sur les modèles la caractérisant. Réaliser cette synthèse manuellement, même à partir d'ensembles de données de petite taille, est fastidieux et faillible. De plus, nous avons précédemment discuté des défis relatifs à la réussite de cette tâche, ainsi que des problématiques liées à la sémantique ontologique de ces modèles (voir Section 2.4) qui rendent difficile l'automatisation de cette opération.

Dans cette section, nous étudions les approches de synthèse de FMs de la littérature (Section 4.2.1). Nous montrons ensuite comment l'AFC peut encadrer cette synthèse et discutons de ses avantages et inconvénients par rapport aux autres approches, en illustrant la méthode proposée sur un exemple tiré de données réelles (Section 4.2.2 et Section 4.2.3). Enfin, nous évaluons notre méthode sur des données de `SPLOT` (Section 4.2.4).

4.2.1 État de l'art

Deux types d'approches de synthèse, i.e., automatique et semi-automatique, ont fait leur apparition dans la littérature du domaine. Nous les détaillons ci-après.

Synthèse automatique

Synthèse automatique de représentations intermédiaires. Nous définissons des représentations de la variabilité comme "intermédiaires" lorsqu'elles sont à mi-chemin entre l'ensemble extensionnel des descriptions de variantes et la représentation intensionnelle et ontologique des FMs. Leur but est de représenter, par une notation graphique proche de ces modèles, l'ensemble des informations sur la variabilité pouvant être extraites depuis des descriptions. Ces structures intermédiaires ne respectent donc pas les contraintes de formatage des FMs de FODA [KCH+90] : c'est pour cela qu'elles incluent en général toutes les hiérarchies et tous les groupes.

Czarnecki and Wasowski [CW07] proposent une méthode entièrement automatique pour modéliser la variabilité d'une formule propositionnelle. Les modèles extraits sont différents de la notation de FODA [KCH+90] et correspondent à une notation "généralisée" dans laquelle les groupes peuvent se chevaucher et où la hiérarchie de caractéristiques peut être un graphe orienté acyclique. La synthèse est déterministe, car cette notation particulière du modèle ne nécessite pas de faire des choix concernant la hiérarchie et les groupes pour respecter la notation traditionnelle. Cette méthode repose sur des diagrammes de décisions binaires [Bry86] pour extraire un graphe d'implications binaires et identifier les groupes de caractéristiques. She et al. [SRA+14] complètent cette méthode par l'identification des

mutex. En plus du graphe d'implications binaires, ils calculent un graphe des mutex, dont les nœuds représentent les caractéristiques et les arêtes les relations d'exclusions mutuelles. Ils représentent la sémantique logique ainsi extraite sous la forme d'un *graphe de caractéristiques*, qui est une représentation symbolique de tous les modèles compatibles avec la formule propositionnelle initiale. Le graphe de caractéristiques montre la hiérarchie de caractéristiques sous forme de graphe orienté acyclique, met en évidence les caractéristiques cooccurrentes et les mutex. Enfin, les auteurs exposent l'ensemble des groupes de caractéristiques (pouvant se chevaucher) textuellement en complément de la hiérarchie. La dérivation de FMs depuis ces représentations intermédiaires n'est pas étudiée dans ces deux articles. La partie gauche de la Figure 4.1 représente le modèle suivant la notation généralisée extrait de l'exemple présenté en Section 3.2 et la partie droite le graphe de caractéristiques comprenant des mutex, tiré du même exemple. On peut mettre ces structures facilement en correspondance avec l'AOC-poset de la Figure 3.3.

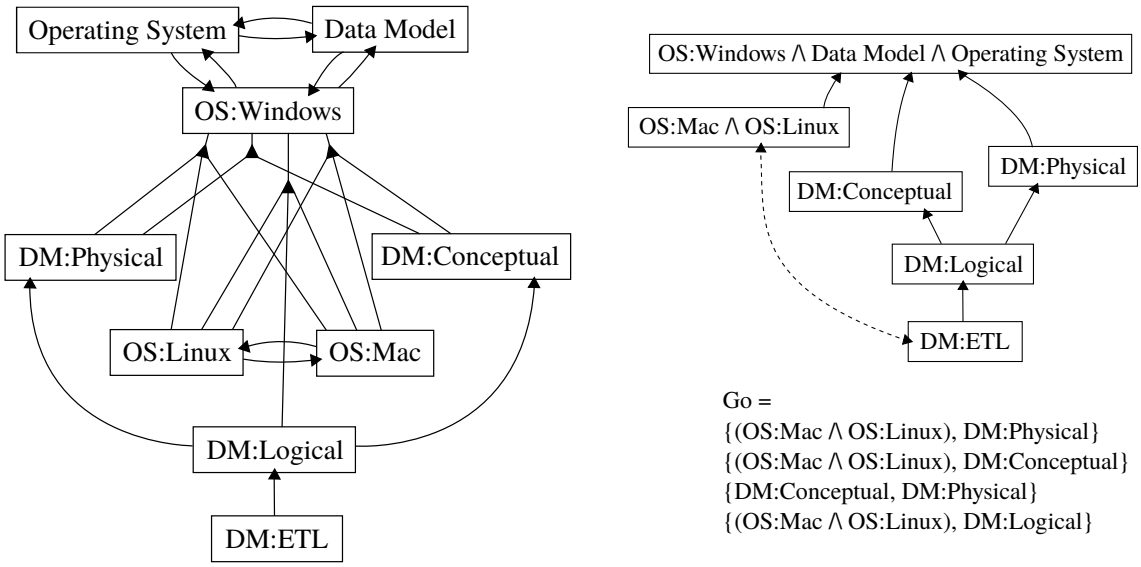


FIGURE 4.1 – (Gauche) Notation généralisée (ancêtre du graphe de caractéristiques) de [CW07] ; (Droite) Graphe de caractéristiques avec mutex de [SRA+14]

Synthèse automatique de FMs. Les travaux de Haslinger et al. [HLE11] cherchent à synthétiser automatiquement un FM depuis une collection d'ensembles de caractéristiques, sans passer par une représentation intermédiaire. La méthode proposée construit d'abord la base du modèle, en identifiant une racine et les caractéristiques présentes dans toutes les variantes. Puis, le reste du modèle est généré récursivement du haut vers le bas : pour chaque caractéristique de la hiérarchie déjà construite, ses descendants directs sont calculés, puis des relations *xor*, *or* et optionnelles sont recherchées parmi eux. Le processus est entièrement automatique et est étendu aux contraintes transverses dans [HLE13]. Certains auteurs utilisent des méta-heuristiques pour obtenir des modèles à partir de configurations. Dans [LLE14], Linsbauer et al. utilisent la programmation génétique pour réaliser cette synthèse, où la fonction objectif sélectionne les modèles ayant une sémantique de configurations la plus proche de l'ensemble initial. Lopez-Herrejon et al. [LLG+15] proposent deux fonctions objectifs pour évaluer les modèles générés : la première cherche à maximiser le nombre de configurations initiales contenues dans le modèle obtenu et la seconde cherche à minimiser le nombre de configurations du modèle généré qui ne sont pas dans l'ensemble initial. L'AFC a déjà été utilisée pour encadrer la synthèse de FMs. La méthode de Ryssel et al. [RPK11] repose sur l'AC-poset pour extraire un arbre de caractéristiques. Ils calculent ensuite les groupes et les mutex pour compléter la hiérarchie

précédemment obtenue. Al-Msie'deen et al. [AMHS+14], utilisent l'AFC comme support à l'extraction de la sémantique logique du modèle et construisent une hiérarchie sur deux niveaux (i.e., la racine puis toutes les autres caractéristiques). Toutes ces approches ont un point commun : la synthèse ne tient pas compte de la sémantique ontologique du modèle final.

Davril et al. [DDH+13] présentent une méthode de synthèse automatique, qui est basée à la fois sur des descriptions de variantes et sur des corpus de textes représentant des descriptions de caractéristiques. Ils cherchent d'abord les ensembles d'items fréquents pour extraire des implications binaires et ainsi construire un graphe d'implications binaires. Puis, ils appliquent des techniques de fouille de texte sur les descriptions de caractéristiques pour tenter d'identifier une hiérarchie cohérente. Cela permet d'approximer la sémantique ontologique des modèles synthétisés. Les résultats ainsi obtenus sont meilleurs que les méthodes automatiques précédentes, mais dépendent fortement des seuils passés en paramètres lors de la recherche d'items fréquents et du processus de fouille de texte.

Les méthodes automatiques sont efficaces pour extraire des structures intermédiaires incluant tous les choix ontologiques qui peuvent être faits pour en dériver un modèle respectant la notation FODA. Elles ne nécessitent donc pas de choix ontologiques lors de leur construction, contrairement aux FMs. Dériver automatiquement un FM nécessite une attention particulière concernant la cohérence de sa hiérarchie lors des choix ontologiques réalisés. Les techniques de traitement automatique du langage naturel semblent être un premier pas vers l'automatisation éventuelle de cette partie du processus. Les méthodes de synthèse automatique ne prenant pas en compte cet aspect ontologique produisent le plus souvent des FMs incohérents.

Synthèse semi-automatique

Un certain nombre de travaux tablent sur la synthèse semi-automatique dans le but d'obtenir une hiérarchie cohérente en prenant en compte les choix d'un utilisateur dans le processus.

She et al. [SLB+11] complètent les travaux de Czarnecki et Wasowski [CW07] en guidant l'utilisateur dans la dérivation d'un modèle conforme à la notation FODA depuis la notation généralisée. Pour cela, ils considèrent, en plus de la formule propositionnelle initiale, un corpus de textes décrivant les caractéristiques. Les auteurs proposent une heuristique basée sur le corpus, identifiant les meilleurs candidats pour être le parent d'une caractéristique donnée, aidant ainsi l'utilisateur dans le choix d'une hiérarchie. Le choix des groupes à retenir dans le modèle final dépend aussi de l'utilisateur, mais aucune heuristique ne propose de classement des groupes par pertinence.

Acher et al. [ACP+12] cherchent à extraire des FMs depuis des descriptions de variantes et des choix utilisateurs. Ils construisent une hiérarchie de caractéristiques finale en produisant une hiérarchie pour chaque variante et en les fusionnant. Chacune de ces hiérarchies est construite en fonction d'une description de la variante et d'une collection de spécifications définie par un concepteur grâce à un langage dédié. Cette paramétrisation permet d'insuffler de la sémantique ontologique dans la hiérarchie. De plus, du fait que différents modèles peuvent être extraits d'un même ensemble de configurations, les auteurs proposent dans [ABH+13] de configurer l'extraction avec un ensemble de propriétés que l'on souhaite voir vérifiées dans le modèle synthétisé. Dans ces deux articles, les décisions de l'utilisateur sont prises avant le début de la procédure de synthèse du modèle.

L'importance du sens de la hiérarchie a été récemment discutée par Becan et al. [BABN16]. Ils proposent une approche hybride basée sur des heuristiques logiques en-

tièrement automatisées et sur des heuristiques ontologiques qui dépendent de décisions de l'utilisateur. Leurs heuristiques logiques extraient les cooccurrences, les groupes de caractéristiques et le graphe d'implications binaires incluant tous les arbres de caractéristiques possibles. Ensuite, leurs heuristiques ontologiques aident l'utilisateur dans le choix 1) d'un arbre de caractéristiques en classant pour chaque caractéristique ses parents potentiels et 2) des groupes à conserver dans le modèle final en détectant si des caractéristiques sont voisines. Ils montrent empiriquement que les méthodes de synthèse hybrides, i.e., qui sont en partie basées sur les décisions d'un utilisateur, surpassent les autres approches en termes de cohérence des modèles synthétisés.

Les méthodes semi-automatiques permettent de 1) construire automatiquement la partie du modèle qui peut être inférée depuis les descriptions de variantes et 2) de solliciter l'utilisateur uniquement lorsque cela est nécessaire, limitant ainsi les erreurs durant la construction du modèle final tout en assurant la cohérence de sa hiérarchie.

Conclusion

Les nombreux travaux sur la synthèse de FMs aident à préciser la démarche qui semble la plus efficace. Elle se déroule en deux étapes. La première consiste à extraire automatiquement une structure intermédiaire représentant toutes les informations sur la variabilité et incluant tous les choix ontologiques pouvant être faits pour dériver un modèle. La seconde étape cherche à "élaguer" cette structure intermédiaire en fonction des décisions d'un utilisateur qui porteront sur le choix des groupes et de la hiérarchie (i.e., choix ontologiques) jusqu'à l'obtention d'un modèle conforme à la notation FODA.

Les structures de l'AFC sont des structures intermédiaires de la variabilité, dans le sens où elles regroupent l'ensemble des choix de modélisation. L'ensemble des informations de la variabilité est ici contenu dans une seule et même structure et ne nécessite pas la construction de graphes d'implications ni de graphes des mutex pour composer une structure finale. En réalité, ces deux graphes sont naturellement inclus dans les structures conceptuelles de l'AFC. De plus, contrairement à la plupart des approches et des structures proposées, qui sont définies de manière fonctionnelle pour répondre à cette problématique, l'AFC est un cadre structurel qui apparaît naturellement comme un espace de représentation de la variabilité. Pour ces deux raisons, nous étudions l'utilisation des structures conceptuelles pour la première étape de la synthèse semi-automatique de modèles.

4.2.2 Une structure intermédiaire canonique pour représenter la variabilité

Nous avons montré dans le chapitre précédent que les structures conceptuelles incluaient la sémantique logique des FMs ayant la même sémantique de configurations et qu'elles représentaient donc des classes d'équivalence de FMs. Les 5 types de relations logiques correspondant à la sémantique de ces modèles sont cependant parfois difficiles à lire dans les structures de l'AFC (e.g., groupes, mutex). Afin de faciliter la lecture de ces classes d'équivalence, nous décidons de représenter leur sémantique logique de manière diagrammatique, dans une représentation proche de celle des FMs. Cette représentation inclut toutes les relations logiques extraites des structures conceptuelles et représente donc de manière compacte et exacte les classes d'équivalence. Nous appelons l'ensemble de relations logiques obtenu à partir de descriptions de variantes une *classe d'équivalence de diagrammes de caractéristiques* (*Equivalence Class Feature Diagram* en anglais, abrégé en ECFD). La Figure 4.2 représente le méta-modèle des ECFDs.

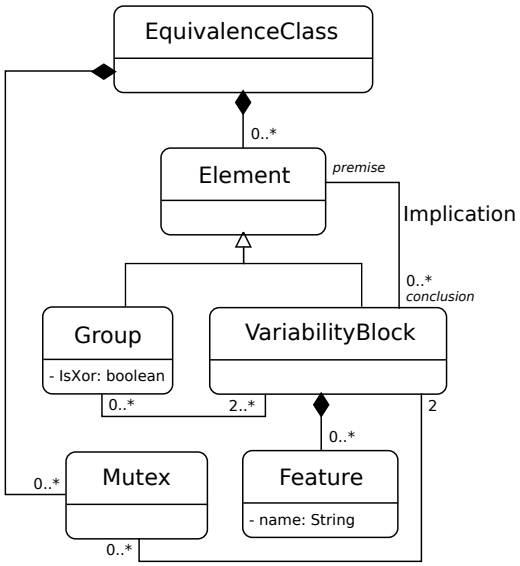


FIGURE 4.2 – Méta-modèle des classes d'équivalence de diagrammes de caractéristiques

Une classe d'équivalence (`EquivalenceClass`) peut posséder deux types d'éléments (`Element`). Un bloc de variabilité (`VariabilityBlock`) groupe un ensemble de caractéristiques (`Features`) cooccurrentes, qui peuvent donc être manipulées comme une seule entité. Un bloc de variabilité est un singleton s'il ne possède qu'une caractéristique et un bloc composite s'il regroupe plusieurs caractéristiques. Un groupe (`Group`) rassemble au moins deux blocs de variabilité et représente un groupe *or* (minimal) ou un groupe *xor*. Une `Implication` depuis un élément (`premise`) vers un bloc de variabilité (`conclusion`) indique que lorsque les caractéristiques du premier élément sont présentes dans une configuration valide, alors les caractéristiques du second élément le sont aussi. Les blocs de variabilité inclus dans un groupe ne peuvent être la `conclusion` de ce groupe. Un groupe est toujours la prémisse d'une implication, c'est-à-dire que tous les blocs d'un groupe impliquent le même bloc de variabilité. De ce fait, tous les blocs de variabilité d'un groupe possède le même parent (i.e., la conclusion de l'implication) : une implication d'un groupe vers un bloc représente une implication de chaque bloc du groupe vers le même bloc parent. Enfin, les exclusions mutuelles (`Mutex`) peuvent être définies entre deux blocs de variabilité. Notons qu'un ECFD n'est pas toujours un arbre et que certains groupes peuvent se chevaucher.

Un ECFD représente donc la sémantique logique d'un ensemble de configurations, à partir des relations extraites des structures conceptuelles. Sa construction est donc déterministe.

Proposition 4.1. *Soit v_c un ensemble de configurations construites sur un ensemble de caractéristiques F. Alors, il existe un unique ECFD, $E = (F_E, VB_E, OR_E, XOR_E, M_E, Imp_E)$ tel que :*

- *(`Features`) $F_E = F$.*
- *(`VariabilityBlock`) VB_E forme une partition de F_E telle que $\forall f_1, f_2 \in F_E$ avec $(\forall\ C \in v_c, f_1 \in C \iff f_2 \in C)$, nous avons $\exists b \in VB_E$ et $f_1 \in b, f_2 \in b$.*
- *(`Group (or)` - minimaux et non purs) $g \in OR_E$ est une paire $(b_0, \{b_1, \ldots, b_n\})$ avec $b_i \in VB_E, i \in \{0, 1, \ldots, n\}$ telle que $\forall\ C \in v_c, b_0 \in C$ si et seulement si $(b_1 \vee \ldots \vee b_n) \in C$ et $\nexists b_k | (b_0, \{b_1, \ldots, b_n\} \setminus b_k) \in OR_E$.*
- *(`Group (xor)`) $g \in XOR_E$ est une paire $(b_0, \{b_1, \ldots, b_n\})$ avec $b_i \in VB_E, i \in \{0, 1, \ldots, n\}$ telle que $\forall\ C \in v_c, b_0 \in C$ si et seulement si $(b_1 \oplus \ldots \oplus b_n) \in C$.*

— *(`Mutex`)* $m \in M_E$ *est une paire* $m = (b_1, b_2), b_1, b_2 \in VB_E$ *telle que* $\forall\, C \in v_c, (b_1 \notin C$ *ou* $b_2 \notin C)$.
— *(`Implication`)* $i \in Imp_E$ *est une paire* $i = (b_1, b_2), b_1, b_2 \in VB_E$ *telle que* $\forall\ C \in v_c, (b_1 \in C$ *implies* $b_2 \in C)$.

Nous définissons une notation textuelle et graphique pour l'ECFD. La notation textuelle rassemble toutes les relations logiques extraites des structures de l'AFC, telle que présentée dans la colonne centrale de la Table 4.1. Elle correspond à la notation de la sémantique logique des FMs telle que nous l'avons présentée en Tables 3.4, 3.5 et 3.6, complétée avec une notation particulière pour les blocs de caractéristiques, permettant de manipuler chaque cooccurrence comme une seule entité. La notation graphique associe à chaque type de relation logique un élément graphique proche de la notation des FMs, tel que présenté dans la colonne de droite de la Table 4.1.

TABLE 4.1 – L'ECFD : un langage de modélisation de la sémantique logique des modèles de variabilité

Élément	**Notation textuelle**	**Notation graphique**
`Feature`	f	f
`Implication`	$f_1 \rightarrow f_2$	f_1 → f_2
`Mutex`	$f_1 \rightarrow \neg f_2$ or $f_2 \rightarrow \neg f_1$	f_1 -x- f_2
`VariabilityBlock` (co-occurrences)	$f_1 \leftrightarrow f_2$	f_1 f_2
`Group (isXor=false)`	$f_0 \rightarrow (f_1 \vee \ldots \vee f_k)$	f_0 ; V ; f_1 ··· f_k
`Group (isXor=true)`	$f_0 \rightarrow (f_1 \oplus \ldots \oplus f_k)$	f_0 ; ⊕ ; f_1 ··· f_k

Exemple. Considérons le contexte formel de la Table 3.1 et son AC-poset associé, à gauche de la Figure 3.4. L'ensemble des relations logiques extraites de cet ensemble de variantes est donné en Figure 4.3. La notation $[f_1, f_2, \ldots, f_n]$ représente un ensemble de caractéristiques $\{f_1, f_2, \ldots, f_n\}$ cooccurrentes. La relation `DM:Conceptual` → `[OS:Windows, DM, OS]` correspond aux trois relations `DM:Conceptual` → `OS:Windows`, `DM:Conceptual` → `DM` et `DM:Conceptual` → `OS`. La notation graphique de cet ECFD est présentée en Figure 4.4. On peut voir que cet ECFD n'est pas un arbre (la caractéristique `DM:Logical` a deux parents) et que plusieurs groupes se chevauchent.

L'ECFD représente une structure intermédiaire de la variabilité, au même titre que la notation généralisée introduite par Czarnecki et Wasowski [CW07] et le diagramme de caractéristiques de She et al. [SLB+11]. La notation généralisée est une représentation de

Blocs composites :	`OS:Linux` $\leftrightarrow$ `OS:Mac`
	`OS:Windows` $\leftrightarrow$ `Data Model (DM)` $\leftrightarrow$ `Operating Systems (OS)`
Implications :	`[OS:Mac,OS:Linux]` $\rightarrow$ `[OS:Windows, DM, OS]`
	`DM:Conceptual` $\rightarrow$ `[OS:Windows, DM, OS]`
	`DM:Physical` $\rightarrow$ `[OS:Windows, DM, OS]`
	`DM:Logical` $\rightarrow$ `DM:Conceptual`
	`DM:Logical` $\rightarrow$ `DM:Physical`
	`DM:ETL` $\rightarrow$ `DM:Logical`
Groupes *or* minimaux :	`[OS:Windows, DM, OS]` $\rightarrow$ (`[OS:Mac,Os:Linux]` $\vee$ `DM:Conceptual`)
	`[OS:Windows, DM, OS]` $\rightarrow$ (`[OS:Mac,Os:Linux]` $\vee$ `DM:Physical`)
	`[OS:Windows, DM, OS]` $\rightarrow$ (`[OS:Mac,Os:Linux]` $\vee$ `DM:Logical`)
	`[OS:Windows, DM, OS]` $\rightarrow$ (`DM:Physical` $\vee$ `DM:Conceptual`)
Goupes *xor* :	$\emptyset$
Mutex :	`[OS:Mac,Os:Linux]` $\rightarrow \neg$ `DM:ETL`

FIGURE 4.3 – Représentation textuelle de l'ECFD extrait de l'AC-poset de la Figure 3.4

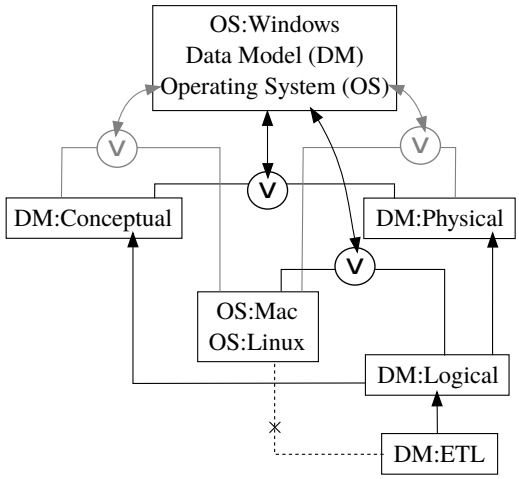


FIGURE 4.4 – Représentation graphique de L'ECFD de la Figure 4.3

la variabilité entièrement graphique mais incomplète, car elle ne montre pas les mutex. De plus, elle ne met pas en évidence les cooccurrences. Cette notation correspond à la réduction transitive du graphe d'implications binaires, dans lequel on ferait apparaître les groupes. Le diagramme de caractéristiques est quant à lui complet, car il reprend la notation généralisée, en y rajoutant les mutex et en faisant apparaître clairement les cooccurrences. Cependant, les groupes n'apparaissent pas dans le diagramme. Faire apparaître les groupes ou non dans le diagramme est en effet un choix discutable, car un nombre trop important de groupes peut complexifier grandement la représentation. Choisir d'afficher certains groupes semble être une stratégie pertinente dans ce cas-là. Ces deux structures sont construites de manière fonctionnelle (i.e., dans l'unique but de représenter la variabilité) et sont basées sur différentes structures e.g., le graphe d'implications binaires et le graphe des mutex. L'ECFD quant à lui, découle directement des structures conceptuelles, dont il est une représentation succincte concentrée sur la variabilité. Il ne contient cependant pas toutes les informations contenues dans le treillis de concepts.

4.2.3 L'ECFD : un guide pour la synthèse de FMs

Nous présentons ici une approche pour synthétiser un modèle depuis un ECFD. Nous montrons les capacités naturelles de l'ECFD pour guider et restreindre les décisions de

l'utilisateur lors de la synthèse, en nous basant sur le mapping mis en place en Section 3.5.

Dériver un modèle depuis un ECFD

Nous avons vu que la sémantique ontologique d'un FM résidait dans le choix d'une hiérarchie cohérente et dans la sélection de groupes de caractéristiques rationnels. Les structures conceptuelles et leurs ECFDs incluent tous les choix possibles et donc l'ensemble des points de variations nécessitant un choix de l'utilisateur. De plus, grâce au mapping établi en Section 3.5 entre les relations logiques mises en évidences dans les structures de l'AFC et les relations ontologiques des FMs, nous sommes capables de présenter l'ensemble des choix possibles pour chaque point de variation de l'ECFD. La résolution de tous les points de variation par l'utilisateur mène à un unique FM. Lorsque le mapping ne présente qu'un seul choix pour un point de variation donné, celui-ci peut être inféré automatiquement. La Figure 4.5 montre un schéma résumant cette méthode de synthèse de modèle, que nous détaillons un peu plus bas.

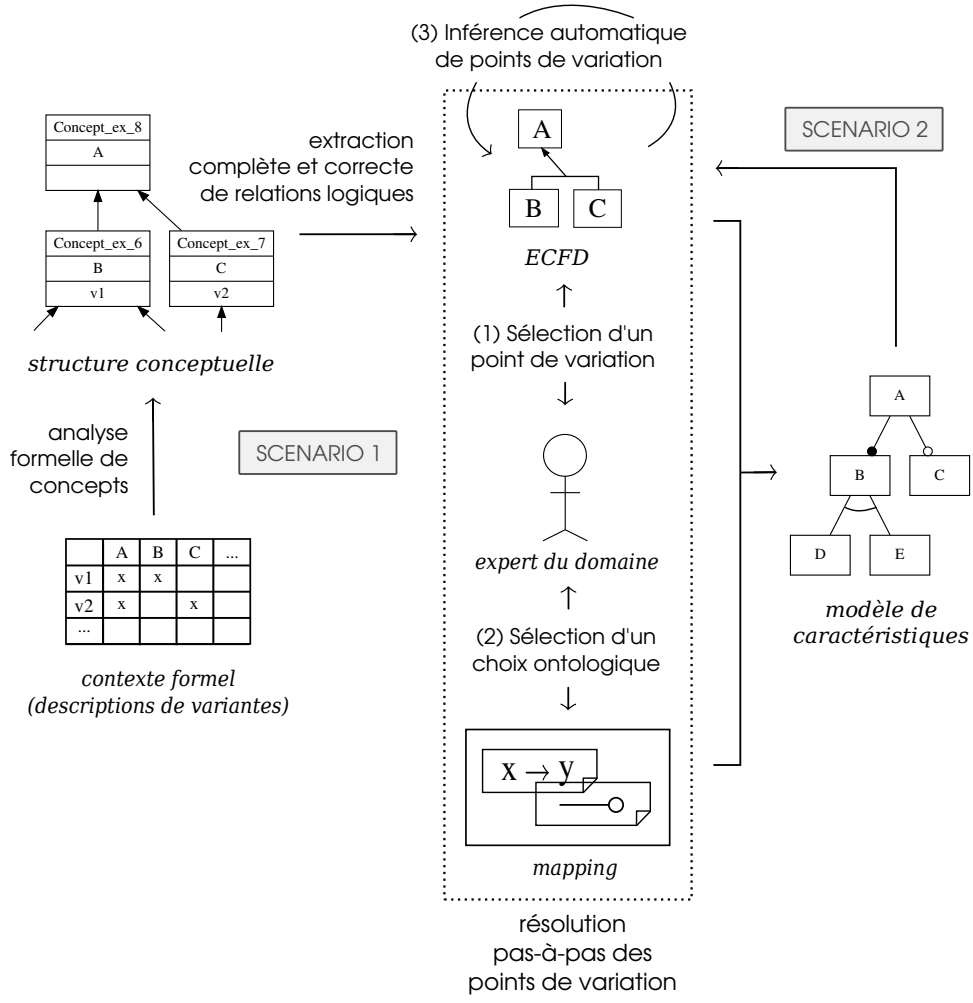


FIGURE 4.5 – Méthode d'assistance à la synthèse d'un FM basée sur la représentation de la variabilité des structures conceptuelles (sous la forme d'ECFD) et du mapping de la Section 3.5

Cette méthode présente plusieurs avantages. Tout d'abord, grâce au mapping restreignant les choix de l'utilisateur, les FMs dérivés ont une sémantique de configurations très proche. Celles-ci ne sont pas toujours équivalentes et peuvent différer légèrement du fait

des choix des groupes par l'utilisateur. Les caractéristiques appartenant aux groupes rejetés par l'utilisateur sont alors automatiquement liées par des relations optionnelles à leur parent, résultant en un ensemble de configurations plus large. Cependant, si plusieurs modèles sont synthétisés à partir du même ensemble de configurations, pour lesquels l'utilisateur sélectionne le même ensemble de groupes, alors leur sémantique de configurations est équivalente, même si la hiérarchie de ces modèles est différente. De plus, le nombre de choix est fortement restreint car (a) la partie ontologique est synthétisée automatiquement, (b) les choix de l'utilisateur sont réduits à ceux du mapping et (c) si un choix utilisateur impacte d'autres points de variation en réduisant leurs choix possibles à exactement 1, il peut être inféré automatiquement. Enfin, le modèle obtenu possède, par construction, une sémantique ontologique validée par l'utilisateur.

On peut voir dans le schéma de la Figure 4.5 deux scénarios d'utilisation. Le premier correspond à la construction de l'ECFD depuis un ensemble de descriptions afin de dériver un FM. Le second scénario correspond à la ré-ingénierie d'un modèle existant. En effet, en dérivant la sémantique logique d'un modèle, on peut obtenir son ECFD correspondant. Dans ce cas-là, aucun groupe n'en chevauche un autre, car cela n'est pas possible dans le modèle depuis lequel l'ECFD est dérivé. De ce fait, un utilisateur peut retravailler la sémantique ontologique d'un FM existant, sans changer sa sémantique de configurations. Loesch et Ploedereder [LP07] ont proposé une méthode de ré-ingénierie de FMs basée sur les treillis de concepts. Pour cela, ils listent l'ensemble des configurations valides d'un modèle sous la forme d'un contexte formel et utilisent les informations mises en évidence dans cette structure (e.g., *dead features*, *core features*) pour proposer des modifications au modèle de départ. Cependant, cette méthode devient difficile à appliquer lorsque le modèle représente un nombre trop important de configurations pour qu'elles puissent être listées et manipulées facilement. En comparaison, notre méthode basée sur l'AFC, l'ECFD et le mapping, ne souffre pas du passage à l'échelle, car le processus ne nécessite pas de lister l'ensemble des configurations [CBHN16].

Autour des points de variation de l'ECFD

Dans ce qui suit, nous étudions les différents points de variation à prendre en compte pour dériver un FM correct depuis un ECFD. Nous illustrons les choix possibles ainsi que la dérivation complète d'un modèle.

La racine de l'arbre des caractéristiques. Traditionnellement, la racine de l'arbre des caractéristiques représente le nom du système modélisé. Si elle existe dans les descriptions de départ, elle se situe dans le *top concept* des structures conceptuelles associées et donc dans le bloc de variabilité de l'ECFD n'ayant pas de parent. Dans notre exemple de la Figure 4.4, trois caractéristiques sont candidates : `OS:Windows`, `Data Model` et `Operating System`. La famille de produits modélisées ici représente des outils de modélisation de données : aucune des trois candidates ne représente donc la racine du modèle. On peut donc choisir d'en ajouter une, que nous décidons d'appeler `DM tools`.

Les groupes de caractéristiques. Une règle à respecter pour dériver un modèle correct est que les groupes retenus doivent être distincts deux à deux. Notre exemple présente 4 groupes qui se chevauchent et on peut trouver deux groupes qui ne se chevauchent pas parmi eux : {`[OS:Mac, OS:Linux], DM:Logical`} et {`DM:Conceptual, DM:Physical`}. Alors que ce dernier semble pertinent, le premier mélange des caractéristiques représentant des fonctionnalités différentes (i.e., des systèmes d'exploitation et des modèles de données). La même remarque s'applique aux deux autres groupes se chevauchant, {`[OS:Mac, OS:Linux], DM:Conceptual`} et {`[OS:Mac, OS:Linux], DM:Physical`}. {`DM:Conceptual,`

DM:Physical} semble être un groupe pertinent. Notons cependant que l'on ne montre ici que les groupes minimaux. Comme nous l'avons vu précédemment en Section 3.4, l'utilisateur peut choisir de compléter un groupe *or* minimal qui lui semble incomplet. Les blocs de variabilité candidats pour compléter le groupe sont les fils de la caractéristique mère du groupe, à savoir DM:Logical et [OS:Mac, OS:Linux]. Nous choisissons de le compléter avec DM:Logical et donc de conserver le groupe {DM:Conceptual, DM:Physical, DM:Logical}.

L'arbre (ou hiérarchie) des caractéristiques. Afin d'obtenir un modèle correct, chaque caractéristique (exceptée la racine) et chaque groupe de caractéristiques doit avoir un seul parent. Le parent correspondant à la caractéristique d'un bloc donné peut être choisi parmi plusieurs blocs. Première possibilité, le parent peut être choisi parmi les caractéristiques d'un des blocs parents, un bloc parent étant la conclusion d'une implication dont la prémisse est le bloc étudié. Dans ce cas-là, les deux caractéristiques sont liées par une relation optionnelle. Seconde possibilité, le parent peut être choisi parmi les caractéristiques introduites dans le même bloc que celle étudiée, si celui-ci n'est pas un singleton. Dans ce cas-là, les deux caractéristiques sont liées par une relation obligatoire, car elles sont cooccurrentes.

Par exemple, la caractéristique DM:ETL est seule dans son bloc ; son parent devra donc être choisi dans un bloc parent. Le seul bloc parent est le singleton DM:Logical : cette caractéristique sera obligatoirement le parent de DM:ETL et elles seront donc reliées par une relation optionnelle. Cette décision peut être inférée automatiquement car aucune autre option n'est permise par le mapping.

Regardons à présent le bloc composite [OS:Mac,OS:Linux]. On peut choisir OS:Mac comme caractéristique parent de OS:Linux : elles seront donc liées par une relation obligatoire car elles sont dans le même bloc. Dans ce cas-là, le parent de OS:Mac ne pourra être qu'une caractéristique d'un bloc parent, car aucune autre caractéristique de son bloc ne peut être désignée pour remplir ce rôle sans invalider l'arbre des caractéristiques. Étant donné qu'il n'existe qu'un seul bloc parent, les candidats pouvant être le parent de OS:Mac dans la hiérarchie sont donc OS:Windows, Data Model et Operating System : le parent sélectionné sera relié à OS:Mac par une relation optionnelle. On peut également faire le contraire : choisir OS:Linux comme parent de OS:Mac et désigner un parent pour OS:Linux parmi les trois caractéristiques du bloc parent. Enfin, une troisième solution est de désigner pour les deux caractéristiques, un parent appartenant au bloc parent. Les deux relations mises en place seront donc optionnelles et la cooccurrence entre OS:Mac et OS:Linux s'exprimera alors par une double implication transverse. Les deux caractéristiques peuvent avoir le même parent, ou bien un parent différent. Nous choisissons cette dernière solution, car nous considérons qu'aucune des deux caractéristiques ne spécialise l'autre et nous désignons le même parent pour les deux : Operating System. La Figure 4.6 montre différents choix possibles (mais non exhaustifs) concernant ce bloc.

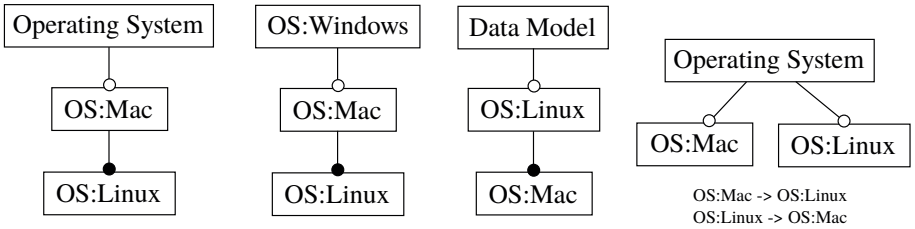


FIGURE 4.6 – Différents choix ontologiques pour le bloc [OS:Mac,OS:Linux]

Considérons maintenant le bloc composite [OS:Windows, Data Model, Operating System]. Il contient en réalité une quatrième caractéristique (DM tool) correspondant à la

racine, que nous avons décidé d'ajouter précédemment. Dans un tel bloc, toutes les combinaisons de relations obligatoires sont possibles, dès lors que `DM tool` reste la racine de l'arbre. Dans notre cas, nous estimons que `OS:Windows` spécialise `Operating System` : nous relions donc (uniquement par des relations obligatoires) `Data Model` et `Operating System` avec `DM tool`, puis `OS:Windows` avec `Operating System`. Il serait possible de relier deux caractéristiques de ce bloc par une relation optionnelle et de rajouter une implication transverse de la fille vers la mère : ce choix ne changerait en effet pas la sémantique de configurations et la sémantique logique. Cependant, nous estimons que c'est un mauvais choix de modélisation (fausse relation optionnelle), qui n'est donc pas proposé dans le mapping. La Figure 4.7 présente différents choix possibles pour ce bloc.

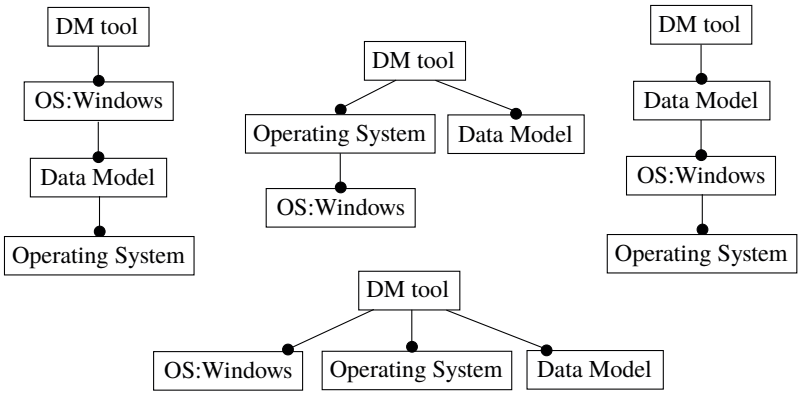
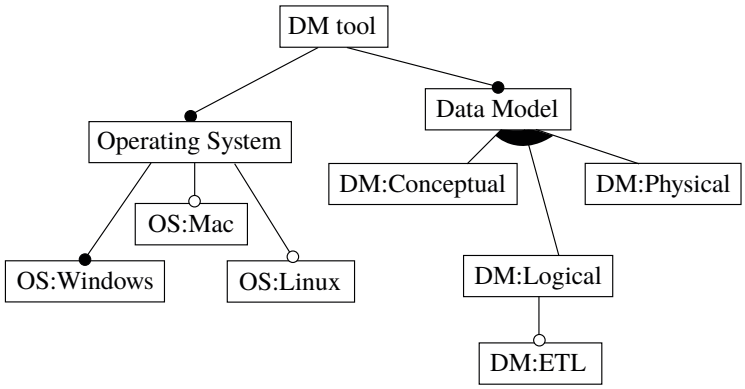


FIGURE 4.7 – Différents choix ontologiques pour le bloc `[OS:Windows, Data Model, Operating System]` et la racine `DM tool`

Le groupe retenu précédemment {`DM:Conceptual, DM:Physical, DM:Logical`} est composé de trois caractéristiques qui doivent avoir le même parent. Dans le cas des groupes, le parent ne peut être choisi que parmi les blocs parents, ce qui réduit les choix. Ici, il faut donc choisir un parent parmi les caractéristiques du bloc majorant et nous désignons `Data Model`.

Enfin, nous pouvons voir que le singleton composé de `DM:Logical` est la prémisse de trois implications. Dans ces cas-là, une seule des trois conclusions peut amener à désigner un parent dans la hiérarchie, les autres devant être des implications transverses afin de conserver la propriété d'arbre. Puisque `DM:Logical` a déjà un parent (i.e., le parent du groupe dans lequel il est inclut), les deux autres implications seront forcément des implications transverses. Ces choix peuvent donc être inférés automatiquement. En effet, si une caractéristique est incluse dans plusieurs implications, dont un groupe choisi par l'utilisateur, alors le groupe prévaut sur les autres implications qui ne peuvent alors plus qu'être traduites en contraintes transverses.

Les points de variation d'un ECFD correspondent donc à : (a) la désignation de la racine, (b) la sélection des groupes et (c) la désignation des parents pour les caractéristiques et les groupes présentant plusieurs choix. Les implications et mutex transverses peuvent être écrits de plusieurs façons différentes, mais nous ne considérons pas que cela représente un point de variation, car cela ne change pas la hiérarchie du modèle. Dans notre exemple, nous avons dû faire 11 choix au total : 1 choix pour la racine, 4 pour les groupes à retenir, 1 pour désigner le parent du groupe retenu et 5 pour désigner les parents des caractéristiques `OS:Windows`, `Operating System`, `Data Model`, `OS:Linux` et `OS:Mac`. Le FM résultant de ces choix est présenté en Figure 4.8.



OS:Mac ↔ OS:Linux ; DM:ETL → ¬ OS:Linux ; DM:ETL → ¬ OS:Mac.

FIGURE 4.8 – FM dérivé de l'ECFD de la Figure 4.4

4.2.4 Évaluation

Nous cherchons à évaluer l'applicabilité et la pertinence de la méthode proposée à travers deux questions de recherche :

QR1 : À quoi ressemblent les ECFDs extraits de données existantes ? Nous pouvons caractériser les ECFDs en fonction des groupes qu'ils possèdent. Puisque nous ne cherchons pas de groupes dits "purs", il est en effet possible qu'un grand nombre de groupes, possiblement non pertinents, soient extraits. Nous savons que certains groupes peuvent se chevaucher, mais ces cas-là sont-ils rares ou au contraire récurrents ? L'ECFD n'est pas forcément un arbre et peut prendre la forme d'un graphe orienté acyclique : qu'en est-il de ces cas-là ? Cette question nous permet aussi d'évaluer l'effort de l'utilisateur lors de la sélection des groupes dans le modèle final.

QR2 : Quel est l'ordre de grandeur du nombre de décisions proposées à un utilisateur grâce à l'ECFD ? En effet, si malgré tout le nombre de décisions devant être prises par l'utilisateur est trop grand pour être praticable, cette méthode sera difficilement applicable.

Données et méthodologie

Afin de pouvoir évaluer correctement la cohérence des ECFDs générés, nous choisissons de réaliser notre évaluation sur des ensembles de configurations dérivés depuis des modèles existants. Ainsi, nous pourrons comparer les relations extraites grâce à notre méthode avec les relations des modèles initiaux. Malgré le fait que nous avons vu que nous pouvions dériver l'ECFD d'un modèle sans avoir à lister son ensemble de configurations valides [CBHN16], nous passons ici par cette étape pour tester les algorithmes d'extraction des relations logiques représentant la variabilité. Nous avons conduit notre évaluation sur des modèles issus de SPLOT, à partir desquels nous construisons l'ECFD basé sur leur ensemble de configurations valides. Le processus utilisé est présenté en Figure 4.9.

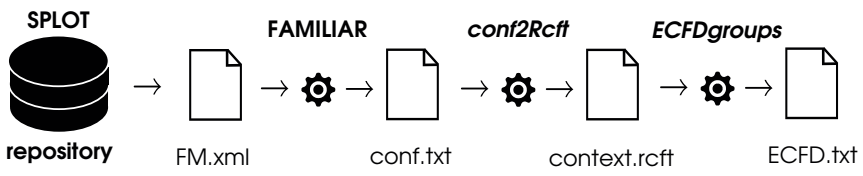


FIGURE 4.9 – Processus de calcul des ECFDs depuis les configurations valides de FMs de SPLOT

Nous utilisons dans ce processus deux outils existants de la littérature : `SPLOT` [MWC09] et `Familiar` [ACLF13]. `SPLOT` propose entre autres un répertoire de plus de 1000 FMs [1] qui ne sont pas d'égale qualité, où chaque modèle peut être récupéré dans un fichier XML. Ces fichiers XML peuvent être pris en charge par `Familiar`, un outil pour l'analyse de FMs. Parmi les opérations proposées, il y en a une qui permet de calculer l'ensemble des configurations valides sous la forme de listes de caractéristiques. Ensuite, nous utilisons deux petits programmes développés pour ce processus. Le premier, *conf2RCFT*, prend en entrée un fichier texte contenant la liste des configurations obtenue avec `Familiar` et la traduit en un contexte formel équivalent au format RCFT. RCFT est un format de fichier représentant des contextes formels, qui sont utilisés par l'outil `RCAExplore` pour construire les structures de l'AFC. Le second programme, *ECFDgroups*, prend un fichier RCFT en entrée et calcule les groupes de caractéristiques et les mutex de l'ECFD.

Nous avons appliqué ce processus sur 10 FMs de `SPLOT`, qui sont présentés dans la Table 4.2. Les modèles sélectionnés possèdent entre 10 et 36 caractéristiques et représentent entre 8 configurations valides (petite LPL) et 8480 configurations valides (large LPL). Pour chaque modèle, nous donnons dans la première partie de la table son nombre de caractéristiques, de configurations, de relations obligatoires et optionnelles, de groupes *or* et *xor* et de contraintes transverses.

QR1 : Pour répondre à cette question, nous calculons les ECFDs et indiquons dans la Table 4.2 le nombre de groupes *or*, *xor* et mutex obtenus. Pour cette évaluation, nous ne consignons que les groupes et mutex, car la complexité de calcul des implications et des cooccurrences est triviale en comparaison. Afin de caractériser au mieux les ECFDs obtenus, nous rapportons aussi le nombre de groupes se chevauchant (non distincts), les cas où les blocs de variabilité possèdent plus d'un parent (`# multi-parents`) et les blocs de variabilité composites (i.e., ayant plus d'une caractéristique). Nous donnons le nombre de blocs composites ainsi que leur taille : (1 :4) dans la table signifie que l'ECFD possède 1 bloc composite ayant 4 caractéristiques.

QR2 : Nous avons vu précédemment que les choix donnés par l'ECFD sont réduits à 1) choisir un parent pour chaque caractéristique lorsque cela est nécessaire et 2) choisir les groupes à garder dans le modèle final. Dans le pire des cas, le nombre de décisions à prendre est donc de :

$$\#\text{ caractéristiques} + \#\text{ or (ECFD)} + \#\text{ xor (ECFD)}$$

Nous donnons de plus une borne inférieure sur le nombre de modèles différents possibles pouvant être dérivés de l'ECFD, afin de les comparer au nombre de décisions laissées à l'utilisateur. Nous ne calculons pas le nombre de choix réels car il est complexe à identifier et que la borne supérieure présentée ci-dessous est suffisante pour discuter de nos résultats. Les blocs de variabilité composites de taille n ($b \in VB_E$ avec $|b| = n$) ont $n(n-1)$ combinaisons possibles de relations obligatoires et $\frac{n(n-1)}{2}$ cas possibles où deux caractéristiques sont optionnelles mais reliées entre elles par une double implication transverse. Le nombre de combinaisons différentes de groupes retenus est égal à 2^m, m étant le nombre de groupes extraits (i.e., $m = |xor|+|or|$). De ce fait, nous considérons la formule suivante pour calculer une borne inférieure du nombre de modèles pouvant être dérivés depuis un ECFD :

$$\prod_{b \in VB_E} (\tfrac{3|b|(|b|-1)}{2}) \times 2^{|xor|+|or|}$$

Nous renseignons dans les deux dernières colonnes de la Table 4.2 la borne supérieure du nombre de décisions présentées par l'ECFD ainsi que la borne inférieure du nombre de modèles dérivables depuis l'ECFD.

1. Dernier accès en juillet 2018

TABLE 4.2 – Informations sur les ECFDs construits depuis les données tirées de SPLOT

Modèles	# caractéristiques	# rel. optionnelles	# rel. obligatoires	# groupes or	# groupes xor	# CTCs	# configurations	# or (ECFD)	# xor (ECFD)	# mutex (ECFD)	# groupes non distincts	# multi-parents	# blocs composites	# choix dans l'ECFD	borne inf. # modèles
Tang Eshop	10	2	3	1	1	2	13	2	1	1	2	1	(1 :4)	13	144
Martini Eshop	11	1	5	1	1	1	8	2	1	1	2	0	(1 :6)	14	360
Toacy Eshop	12	1	3	2	1	0	48	2	1	0	0	0	(1 :4)	15	144
Mobile Games	16	10	1	1	0	1	3645	1	0	0	0	0	(1 :2)	17	6
Web Game	16	5	6	2	0	2	84	3	0	0	0	0	(1 :6)(1 :2)	19	1080
Smart Home	22	5	3	5	0	2	8480	5	0	0	2	1	(1 :2)(1 :3)	27	864
Bicycle	27	6	5	0	5	2	1152	1	5	14	0	0	(1 :6)	33	2880
Automotive system	31	3	8	1	7	9	1344	9	7	9	8	1	(1 :7)(2 :2)	47	1.2E7
Robot Calibration	33	0	10	1	7	11	648	2	7	3	3	0	(1 :14)	42	1.4E5
Online-book-shopping	36	2	21	0	5	3	90	1	4	1	0	0	(1 :18)(2 :4)	41	2.6E5

Analyse des résultats

QR 1 : Il est à noter dans un premier temps que tous les ECFDs ont pu être calculés malgré la complexité de l'algorithme d'extraction des groupes et la taille des ensembles de configurations évalués.

Les résultats montrent que le nombre de groupes peut varier entre l'ECFD et le modèle initial. Par exemple, un groupe *xor* de l'ECFD peut combiner plusieurs groupes *xor* du modèle lorsqu'il y a des contraintes transverses additionnelles. C'est le cas de *Online-book-shopping*, duquel on extrait 4 groupes *xor* au lieu des 5 présents dans le modèle initial. *Smart Home* présente un autre cas intéressant : nous extrayons le même nombre de groupes *or* que dans le modèle initial, mais nous détectons que deux d'entre eux se chevauchent. De ce fait, l'utilisateur ne peut en sélectionner qu'au plus 4. Il est possible que les contraintes transverses incluant des caractéristiques appartenant à des groupes "cassent" la sémantique logique des dits groupes, empêchant ainsi leur détection avec les algorithmes proposés.

Nous pouvons voir que 3 des 10 modèles étudiés possèdent un cas de multi-parenté, i.e., lorsqu'un bloc de variabilité a plusieurs parents. Cependant, cela semble occasionnel. Les groupes qui se chevauchent semblent plus communs, mais pas très nombreux : en moyenne, 30% des groupes extraits se chevauchent. Il arrive parfois qu'un nombre plus conséquent de groupes se chevauchent, par exemple pour *Automotive system*, pour lequel 8 groupes se chevauchent sur les 16 extraits. Des heuristiques permettant de proposer des ensembles maximaux de groupes distincts deux à deux à l'utilisateur pourraient aider l'utilisateur dans le choix des groupes à retenir.

QR 2 : La comparaison entre le nombre de choix proposés par un ECFD et le nombre de modèles dérivables montre à quel point l'utilisation de l'ECFD permet de réduire les décisions de l'utilisateur. Notons aussi que la borne supérieure du nombre de choix est très grande. En effet, nous avons vu que lorsque le bloc parent est un singleton, l'utilisateur n'a pas besoin de prendre de décision. Or, les blocs singletons représentent la majorité des blocs de variabilité : dans notre expérimentation, 87.5% des blocs de variabilité extraits

sont des singletons. Le nombre de choix réels est donc bien inférieur à celui présenté ici.

En résumé (4.2) :

- La **synthèse de FMs** consiste à extraire un modèle cohérent depuis des descriptions d'une FPL.
- Les approches les plus efficaces sont **semi-automatiques**, où une **structure intermédiaire** représentant tous les choix ontologiques est extraite automatiquement, de laquelle un modèle est dérivé grâce aux **décisions de l'utilisateur**.
- Les structures conceptuelles sont des structure intermédiaires obtenues de manière structurelle et qui incluent toutes les informations de variabilité nécessaires à la synthèse de FMs.
- L'**ECFD** est une représentation simplifiée des structures conceptuelles qui est **centrée sur la variabilité**, qui permet de restreindre et de guider l'utilisateur dans ses décisions lors de la synthèse.
- L'ECFD permet, pour un sous-ensemble de groupes fixé, de dériver des modèles ayant la **même sémantique de configuration**.

4.3 Union et intersection de modèles de FPLs

Lorsque l'on travaille sur une LPL assez grande, il est inenvisageable de gérer un unique FM, ayant potentiellement une taille conséquente et représentant l'intégralité de la variabilité de la LPL. C'est d'ailleurs le credo de l'approche OVM (pour *Orthogonal Variability Model*) [PBvdL05]. Une solution pour faciliter la gestion de la variabilité dans ces cas-ci consiste à diviser la LPL en fonction d'un certain nombre de préoccupations et à manipuler un modèle plus spécifique pour chacune d'entre elles. Cependant, même si gérer plusieurs petits modèles distincts se révèle plus pratique, il peut être utile pour certaines activités de conception de pouvoir les combiner. L'analyse des points communs entre différentes préoccupations en est un exemple. Définir des opérations permettant la composition des FMs est alors nécessaire [ACLF10b]. Dans le contexte de la ré-ingénierie des LPLs, ces opérations de composition peuvent servir d'autres buts, comme la réutilisation et l'adaptation de FMs. Elles facilitent aussi la mise en place itérative de la migration vers des LPLs (approche réactive [Kru02])à travers l'agrégation de différentes préoccupations ou l'analyse de l'intégration de nouveaux éléments.

Deux principales opérations émergent de la littérature, l'union et l'intersection de modèles et plusieurs approches pour mettre en œuvre ces opérations ont été proposées [ACLF10a]. Les stratégies prévalentes opèrent directement sur les structures des FMs, ou alors sur leurs formules propositionnelles associées. Bien que ces approches aient leurs avantages, soit elles ont tendance à confiner le concepteur dans une vue ontologique prédéfinie, soit elles produisent des résultats approximatifs, soit elles nécessitent un travail significatif pour synthétiser un FM depuis une formule propositionnelle. De plus, la mise en œuvre des opérations intervenant directement sur la structure du modèle prend difficilement en compte les contraintes transverses.

Dans cette section, nous présentons d'abord les différentes opérations de composition et étudions les implémentations existantes de ces opérations (Section 4.3.1). Nous proposons ensuite une implémentation originale basée sur la sémantique de configurations des modèles, permettant de contourner certaines limites des approches existantes. Cette méthode est basée sur l'AFC et ses structures conceptuelles et produit un ECFD représentant la composition de familles de logiciels ayant ou non des modèles de variabilité (Section 4.3.2).

Enfin, nous proposons une nouvelle opération de composition dont la mise en œuvre est rendue possible par l'AFC. Cette opération est basée sur l'étude de configurations partielles communes à deux modèles et est appelée intersection approximative (Section 4.3.3).

4.3.1 Opérations de composition des FMs

Dans ce qui suit, nous définissons l'union et l'intersection des FMs, puis nous discutons des approches existantes implémentant ces deux opérations de composition.

Définitions

Parmi les nombreuses opérations de composition introduites dans [ACLF10b], l'union et l'intersection ont une place spéciale dans la manipulation des modèles afin de donner des vues différentes d'un système. L'union représente une vue intégrée, alors que l'intersection met en évidence le cœur commun des modèles étudiés. Ces deux opérations sont définies en fonction de la sémantique de configurations des modèles, où la notation $[\![FM]\!]$ donne l'ensemble des configurations valides d'un FM :

Définition 4.2 (Intersection [ACLF10a])**.**
L'opération d'intersection, dénotée $\cap$*, construit un FM* FM_3 *à partir de deux modèles* FM_1 *et* FM_2 *tel que* $[\![FM_3]\!]=[\![FM_1]\!] \cap [\![FM_2]\!]$*.*

Définition 4.3 (Union [ACLF10a])**.**
L'opération d'union, dénotée $\cup^{\sim}$*, construit un FM* FM_3 *à partir de deux modèles* FM_1 *et* FM_2 *tel que* $[\![FM_3]\!] \supseteq [\![FM_1]\!] \cup [\![FM_2]\!]$*. Cette union est dite approximative.*

Définition 4.4 (Union stricte [ACLF10a])**.**
L'opération d'union stricte, dénotée $\cup$*, construit un FM* FM_3 *à partir de deux modèles* FM_1 *et* FM_2 *tel que* $[\![FM_3]\!]=[\![FM_1]\!] \cup [\![FM_2]\!]$*.*

Par définition, l'opération d'union stricte est une restriction de l'opération d'union. La Figure 4.12 illustre une intersection et une union stricte sur un exemple simple, avec $[\![F_1]\!]= \{\{A, B\}, \{A, B, C\}\}$ et $[\![F_2]\!]= \{\{A, B\}, \{A, B, D\}\}$. Leurs intersection et union stricte donnent respectivement : $[\![F_1]\!] \cap [\![F_2]\!]= \{\{A, B\}\}$ et $[\![F_1]\!] \cup [\![F_2]\!]= \{\{A, B\}, \{A, B, C\}, \{A, B, D\}\}$. Un exemple plus complet est donné dans la sous-section suivante.

Comparaison des implémentations des opérations de composition

Plusieurs implémentations ont été proposées pour l'union et l'intersection, les deux plus répandues étant basées sur la structure de l'arbre des caractéristiques des modèles (i.e., la sémantique ontologique), ou sur les formules propositionnelles associées (i.e., la sémantique logique) [ACLF10b]. Ces deux approches prennent en entrée deux FMs.

L'approche basée sur la sémantique logique consiste à utiliser les formules logiques équivalentes aux deux modèles à combiner, afin de produire une troisième formule caractérisant le modèle obtenu après la composition. Dans notre cas, la formule pour F_1 peut être $(A \wedge B) \vee (A \wedge B \wedge C)$ et la formule pour F_2 peut être $(A \wedge B) \vee (A \wedge B \wedge D)$. Dans [ACLF10b], la formule proposée pour caractériser l'intersection (resp. l'union stricte) est donnée par l'Équation 4.1 (resp. l'Équation 4.2). Cette approche est complète et correcte et peut être implémentée en utilisant les modèles de départ pour dériver les formules propositionnelles, tel que défini par Batory et al. [Bat05]. Cependant, elle nécessite d'être complétée par une seconde étape de synthèse de FM à partir de la formule obtenue.

$$(((A \wedge B) \vee (A \wedge B \wedge C)) \wedge (\neg D)) \wedge (((A \wedge B) \vee (A \wedge B \wedge D)) \wedge (\neg C)) \tag{4.1}$$

$$(((A \wedge B) \vee (A \wedge B \wedge C)) \wedge (\neg D)) \vee (((A \wedge B) \vee (A \wedge B \wedge D)) \wedge (\neg C)) \quad (4.2)$$

Afin d'étudier l'approche basée sur la sémantique ontologique, nous avons besoin d'introduire un exemple un peu plus complexe et nous utilisons alors une variante de l'exemple sur les applications de e-commerce. La Figure 4.10 présente deux modèles candidats à la composition, qui représentent des applications de e-commerce. La Table 4.3 présente leurs ensembles de configurations valides respectifs. Ces configurations se voient ici attribuer un identifiant unique (e.g., FM_1C_1 est une configuration (C_1) de FM_1).

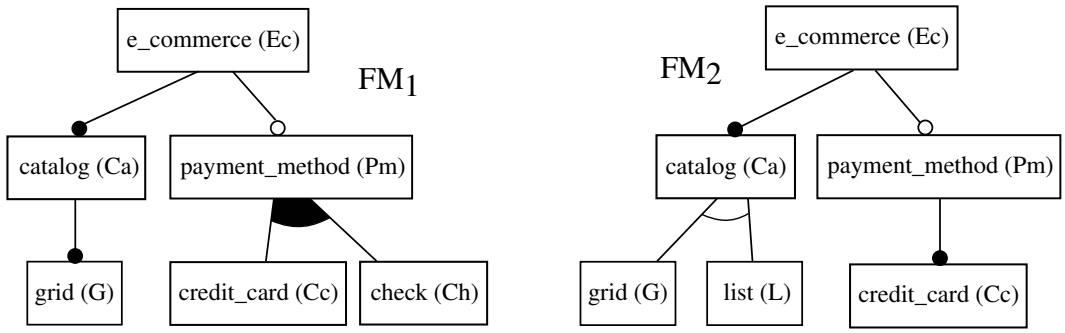


FIGURE 4.10 – Deux FMs (à gauche, FM_1 et à droite, FM_2) représentant deux ensembles d'applications de e-commerces

TABLE 4.3 – Configurations valides de FM_1 et FM_2 de la Figure 4.10.

	Ec	**Ca**	**G**	**Pm**	**Cc**	**Ch**
FM_1C_1	x	x	x			
FM_1C_2	x	x	x	x	x	
FM_1C_3	x	x	x	x		x
FM_1C_4	x	x	x	x	x	x

	Ec	**Ca**	**G**	**L**	**Pm**	**Cc**
FM_2C_1	x	x	x			
FM_2C_2	x	x		x		
FM_2C_3	x	x	x		x	x
FM_2C_4	x	x		x	x	x

Cette approche est basée sur un ensemble de règles de composition, qui sont listées dans [ACLF10b]. Le résultat de l'application de ces règles sur les deux modèles précédents est donné en Figure 4.11. Par exemple, une règle pour l'union définit que la composition du groupe *xor* sous la caractéristique `catalog` de FM_2 et de la relation obligatoire de `catalog` vers `grid` de FM_1 produit un groupe *or* sous `catalog` (voir la partie de droite de la Figure 4.11).

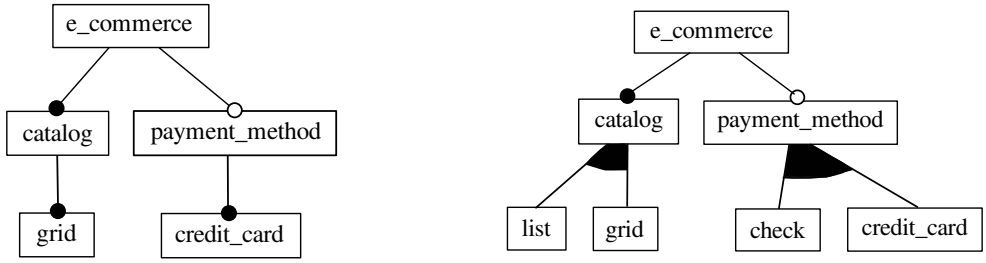


FIGURE 4.11 – Intersection (gauche) et union (droite) de FM_1 et FM_2 de la Figure 4.10 basées sur les règles de composition de [ACLF10b]

Une hypothèse sous-jacente dont dépend l'applicabilité de cette approche est que le même ensemble de caractéristiques soit partagé par les deux modèles à combiner. Ce n'est pas le cas dans notre exemple et ces règles produisent parfois une union non stricte. Par

exemple, la configuration $\{Ec, Ca, G, L, Pm, Ch\}$ n'est pas dans l'union stricte des configurations listées dans la Table 4.3 : en effet, cette configuration possède à la fois L qui n'est pas dans FM_1 et Ch qui n'est pas dans FM_2. Une des propriétés de cette approche est que les règles ne reconsidèrent pas toute la sémantique ontologique et plus particulièrement les relations parent-enfant. Dans notre exemple, deux solutions peuvent être considérées concernant la place de `payment_method` dans la hiérarchie : elle peut être placée (1) en dessous de `e_commerce`, qui est le choix préservé par les règles de composition, ou bien (2) en dessous de `catalog`, qui est une alternative pouvant être considérée par un concepteur et qui ne change pas la sémantique de configurations du modèle obtenu. Cette dernière solution n'est cependant pas proposée par les règles de compositions. De plus, il est important de souligner que cette approche ne prend pas en compte les contraintes transverses, si celles-ci existent.

C'est pourquoi, malgré les qualités de ces approches, il est intéressant d'avoir un point de vue complémentaire basé sur l'ensemble des configurations, qui :

— assure que les opérations d'union stricte et d'intersection sont correctes et complètes ;
— soit capable de prendre en compte les contraintes transverses ;
— puisse traiter des modèles ayant des ensembles de caractéristiques différents ;
— ne confine pas le concepteur dans une vue ontologique spécifique, tout en l'assistant dans la construction d'un modèle cohérent.

Dans ce qui suit, nous proposons une telle solution, qui de plus ne nécessite pas de prendre des modèles bien formés en entrée.

4.3.2 L'AFC pour la composition de familles de logiciels

Aperçu de la méthode proposée

La Figure 4.12 illustre l'approche proposée pour mettre en œuvre les opérations de composition. Les données d'entrée peuvent être : deux FMs (en haut à gauche de la figure), un modèle et un ensemble de configurations (en bas à gauche de la figure), ou deux ensembles de configurations (non illustré ici). L'ensemble des configurations est calculé pour chaque modèle en entrée, qui est représenté sous la forme d'un contexte formel. Dans notre exemple, les modèles F_1 et F_2 (en haut à gauche) ont respectivement FC1 et FC2 comme ensembles de configurations / contextes formels associés. Les opérations de composition (union et intersection) sont ensuite appliquées sur ces contextes formels, pour obtenir un contexte-union et/ou un contexte-intersection, à partir desquels on construit les AC-posets. L'ECFD peut alors être extrait d'un AC-poset, soit pour représenter la variabilité à travers une structure intermédiaire, soit pour assister le concepteur dans la synthèse d'un FM

Contexte-union et contexte-intersection

Cette méthodologie prend en compte des ensembles de configurations valides, représentés sous la forme de contextes formels. Les opérations de composition s'appliquent sur ces contextes formels et prennent la forme d'une union ou d'une intersection ensembliste dont résulte un troisième contexte formel. Tous les types de données à partir desquels on peut extraire un ensemble de configurations peuvent donc être pris en compte. Cependant, si l'un des ensembles de configurations est trop important pour être manipulé efficacement, cela peut nuire à l'applicabilité de cette méthode. Le type de données d'entrée qui a le plus de probabilité de représenter un grand nombre de configurations est le FM. Mais, comme nous supposons ici travailler avec des modèles réduits en fonction de préoccupations spécifiques, cette difficulté est donc limitée.

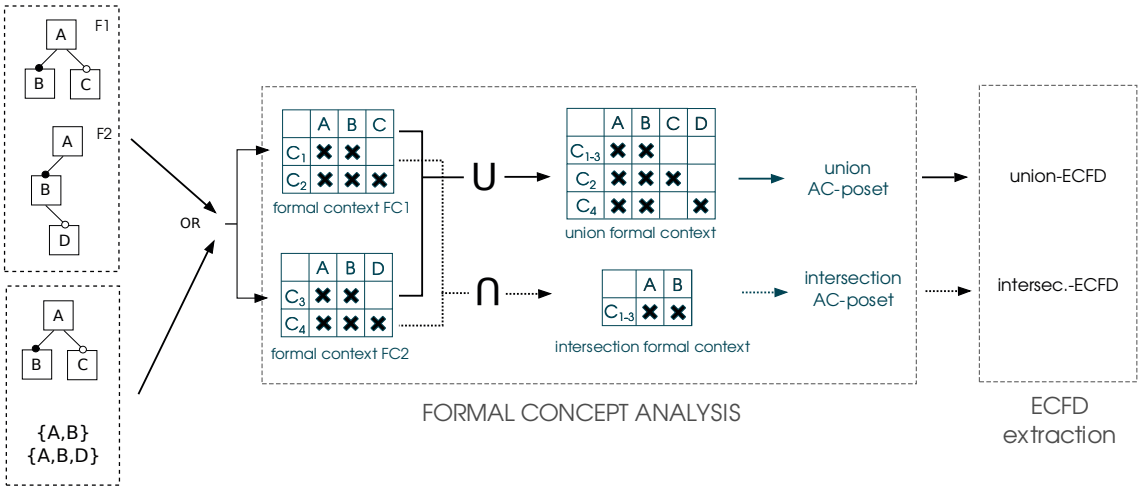


FIGURE 4.12 – Schéma du processus de composition de FPLsbasé sur l'AFC

Pour définir l'union stricte et l'intersection sur les contextes formels, nous introduisons la notion d'égalité des configurations, dénotée $\triangleq$, comme étant des configurations ayant le même ensemble de caractéristiques. Dans les tables et les figures, qui sont générées par des outils, $\triangleq$ sera dénoté par "=".

Dans la définition suivante, pour $K = (O, A, I)$, on note pour $o \in O, I(o) = \{a \in A | (o, a) \in I\}$, c'est-à-dire que par abus de notation, nous considérons la relation $I \subseteq O \times A$ comme une fonction $I : O \mapsto 2^A$.

Définition 4.5 (Égalité des configurations, $\triangleq$)**.** *Soit deux contextes formels* $K_1 = (O_1, A_1, I_1)$ *et* $K_2 = (O_2, A_2, I_2)$. $\forall o_1 \in O_1, \forall o_2 \in O_2$ *:*

$$o_1 \triangleq o_2 \iff I_1(o_1) = I_2(o_2)$$

Nous définissons à présent le contexte formel associé à l'intersection comme l'ensemble des lignes (i.e., configurations), qui sont présentes dans les deux contextes de départ (Définition 4.6). Les lignes du contexte-intersection sont labellisées pour indiquer leur origine. La Table 4.4 montre le contexte formel associé à l'intersection des contextes formels de FM_1 et FM_2, présentés dans la Table 4.3.

Définition 4.6 (Contexte-intersection)**.** *Soit deux contextes formels* $K_1 = (O_1, A_1, I_1)$ *et* $K_2 = (O_2, A_2, I_2)$.

On définit le contexte formel représentant l'intersection $K_1 \cap K_2$ *par* $K_{(K_1 \cap K_2)} = (O_{(K_1 \cap K_2)}, A_{(K_1 \cap K_2)}, I_{(K_1 \cap K_2)})$ *tel que :*

— $O_{(K_1 \cap K_2)} = \{o_{o_1 \triangleq o_2} \mid \exists (o_1, o_2) \in O_1 \times O_2, o_1 \triangleq o_2\}$

— $A_{(K_1 \cap K_2)} = A_1 \cap A_2$

— $I_{(K_1 \cap K_2)} = \{(o_{o_1 \triangleq o_2}, a) \mid a \in A_{(K_1 \cap K_2)}, o_{o_1 \triangleq o_2} \in O_{(K_1 \cap K_2)}, (o_1, a) \in I_1, (o_2, a) \in I_2\}$

TABLE 4.4 – Contexte formel associé avec l'intersection des contextes de la Table 4.3.

	Ec	Ca	G	Pm	Cc
$FM_1C_1 = FM_2C_1$	x	x	x		
$FM_1C_2 = FM_2C_3$	x	x	x	x	x

La Définition 4.7 introduit le contexte formel associé avec l'union stricte de deux contextes formels. L'union étant dans notre implémentation basée sur l'opération ensembliste, les contextes-unions représentent toujours une union stricte : par abus de notation, nous pouvons parfois omettre le qualificatif "stricte" dans la suite de cette section. Le contexte-union obtenu possède les lignes qui sont dans au moins un des deux contextes de départ. Les lignes qui sont communes aux deux contextes (i.e., celles présentes dans le contexte-intersection) sont fusionnées pour n'en former qu'une. La Table 4.5 montre le contexte associé à l'union-stricte des contextes de FM_1 et FM_2 qui sont présentés en Table 4.3. Il met en évidence les deux configurations communes (deux premières lignes) ainsi que les configurations qui sont spécifiques à un modèle (quatre lignes suivantes).

Définition 4.7 (Context-union (stricte))**.** *Soit deux contextes formels* $K_1 = (O_1, A_1, I_1)$ *et* $K_2 = (O_2, A_2, I_2)$*. Considérons :*

- *L'ensemble des configurations communes (de la Définition 4.6)* $O_{(K_1 \cap K_2)}$ *et la relation* $I_{(K_1 \cap K_2)}$
- *L'ensemble des configurations spécifiques à* O_1 *:* $SPE(O_1) = \{o_1 \mid o_1 \in O_1$ *et* $\nexists o_2 \in O_2$*, avec* $o_{o_1 \triangleq o_2} \in O_{(K_1 \cap K_2)}\}$
- *L'ensemble des configurations spécifiques à* O_2 *:* $SPE(O_2) = \{o_2 \mid o_2 \in O_2$ *et* $\nexists o_1 \in O_1$*, avec* $o_{o_1 \triangleq o_2} \in O_{(K_1 \cap K_2)}\}$

On définit le contexte formel représentant l'union $K_1 \cup K_2$ *par* $K_{(K_1 \cup K_2)} = (O_{(K_1 \cup K_2)}, A_{(K_1 \cup K_2)}, I_{(K_1 \cup K_2)})$ *tel que :*

- $O_{(K_1 \cup K_2)} = O_{(K_1 \cap K_2)} \cup SPE(O_1) \cup SPE(O_2)$
- $A_{(K_1 \cup K_2)} = A_1 \cup A_2$
- $I_{(K_1 \cup K_2)} = I_{(K_1 \cap K_2)} \cup \{(o, a) \mid o \in SPE(O_1), a \in A_{(K_1 \cup K_2)}, (o, a) \in I_1\} \cup \{(o, a) \mid o \in SPE(O_2), a \in A_{(K_1 \cup K_2)}, (o, a) \in I_2\}$

TABLE 4.5 – Contexte formel associé à l'union stricte des contextes de la Table 4.3.

	Ec	Ca	G	L	Pm	Cc	Ch
$FM_1C_1 = FM_2C_1$	x	x	x				
$FM_1C_2 = FM_2C_3$	x	x	x		x	x	
FM_1C_3	x	x	x		x		x
FM_1C_4	x	x	x		x	x	x
FM_2C_2	x	x		x			
FM_2C_4	x	x		x	x	x	

L'ECFD, une classe d'équivalence des FMs représentant l'union et l'intersection

À partir des contextes formels associés à l'union stricte et à l'intersection, on peut construire automatiquement les AC-posets correspondant. Les deux ECFDs associés au contexte-union (stricte) de la Table 4.5 et au contexte-intersection de la Table 4.4 sont présentés respectivement à gauche et à droite de la Figure 4.13.

Générer l'ECFD correspondant à l'opération de composition permet de reconsidérer toute la sémantique ontologique du modèle représentant l'union ou l'intersection. Ainsi, le concepteur n'est pas limité à un point de vue ontologique. De plus, l'ECFD permet de dériver des modèles ayant des contraintes transverses. Par exemple, les Figures 4.14 et 4.15 montrent (à gauche) l'ECFD-union annoté avec les décisions d'un concepteur et (à droite) le FM correspondant.

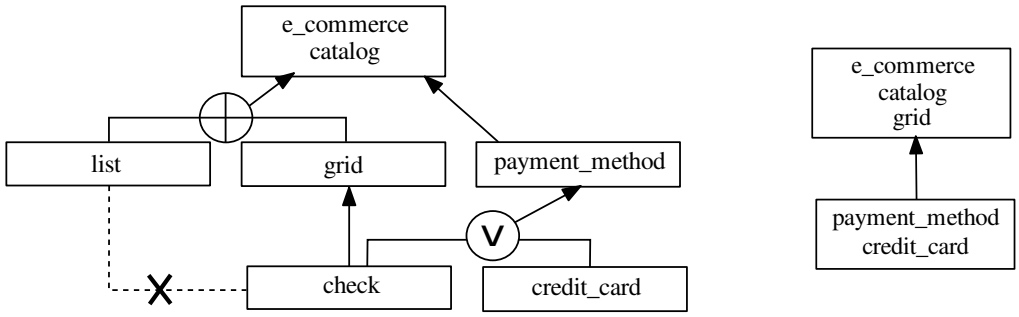


FIGURE 4.13 – ECFDs associés au contexte-union (à gauche) et au contexte-intersection (à droite) des Tables 4.5 et 4.4

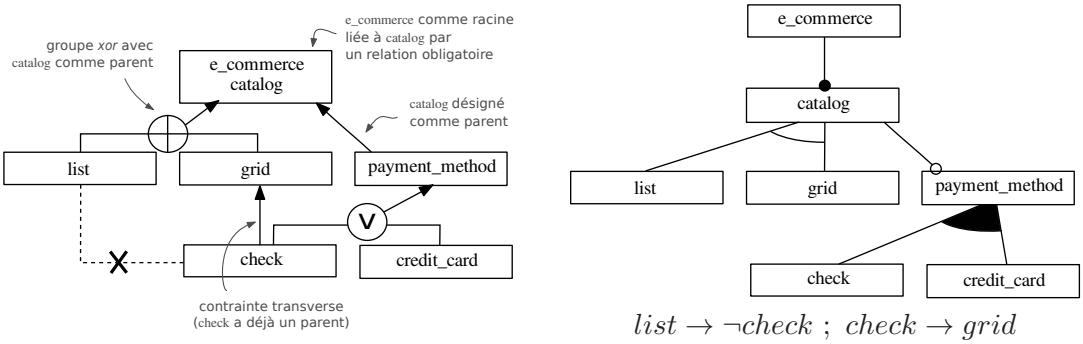


FIGURE 4.14 – ECFDs associés au contexte-union (à gauche) et au contexte-intersection (à droite) des Tables 4.5 et 4.4

Cependant, même si la méthode permet de prendre en compte des ensembles de caractéristiques différents, la présence/absence de caractéristiques dans un ensemble de configurations et pas dans l'autre pose des problèmes pour l'intersection que nous abordons grâce à l'intersection approximative.

4.3.3 L'intersection approximative

Alors que l'union de modèles est toujours informative, l'intersection est souvent vide même si les modèles de départ présentent de fortes similitudes. Nous illustrons ce problème avec une légère modification sur l'exemple des applications de e-commerce de la Figure 4.10. Au modèle FM_1, nous rajoutons une caractéristique `user_management (Um)` sous la caractéristique racine `e_commerce (Ec)` et nous les lions par une relation obligatoire. Le nouveau modèle est dénoté FM_1' est présenté à gauche de la Figure 4.16. Au modèle FM_2, nous ajoutons la caractéristique `paypal (Pp)` à `credit_card (Cc)` par une relation obligatoire. Le modèle modifié, dénoté FM_2', est présenté à droite de la Figure 4.16. Après ces modifications, il n'y a plus de configuration commune aux deux modèles FM_1' et FM_2' : le contexte formel associé à leur intersection est donc vide. La Table 4.6 montre les contexte formels représentant les ensembles de configurations valides de FM_1' et FM_2' et le contexte formel associé à l'union (stricte) de ces deux modèles. L'AC-poset du dernier contexte est présenté en Figure 4.17.

Les concepts dans l'AC-poset mettent en évidence différents types d'informations relatives aux parties communes des deux modèles. Leur étude nous permet de déterminer un cœur commun aux combinaisons de caractéristiques et de catégoriser les configurations partielles. Nous avons identifié deux types de configurations partielles.

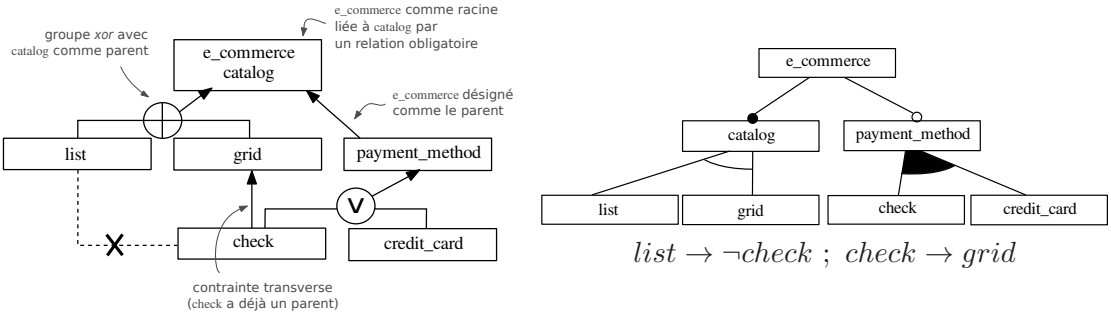


FIGURE 4.15 – ECFDs associés au contexte-union (à gauche) et au contexte-intersection (à droite) des Tables 4.5 et 4.4

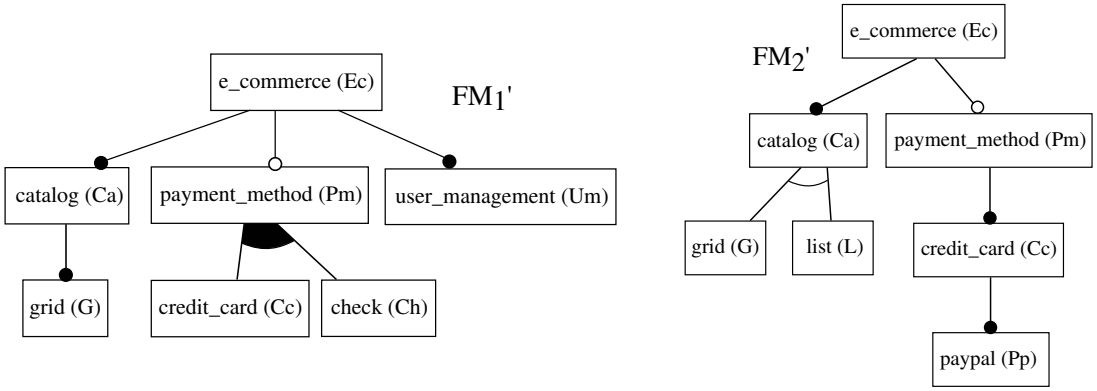


FIGURE 4.16 – Modifications des modèles de la Figure 4.10

TABLE 4.6 – (Haut) : Ensemble de configurations de FM'_1 et FM'_2 de la Figure 4.16. (Bas) : Contexte formel de $FM'_1 \cup FM'_2$.

FM'_1	Ec	Ca	G	Pm	Cc	Ch	Um
FM'_1C_1	x	x	x				x
FM'_1C_2	x	x	x	x	x		x
FM'_1C_3	x	x	x	x		x	x
FM'_1C_4	x	x	x	x	x	x	x

FM'_2	Ec	Ca	G	L	Pm	Cc	Pp
FM'_2C_1	x	x	x				
FM'_2C_2	x	x		x			
FM'_2C_3	x	x	x		x	x	x
FM'_2C_4	x	x		x	x	x	x

$FM'_1 \cup FM'_2$	Ec	Ca	G	L	Pm	Cc	Ch	Um	Pp
FM'_1C1	x	x	x					x	
FM'_1C2	x	x	x		x	x		x	
FM'_1C3	x	x	x		x		x	x	
FM'_1C4	x	x	x		x	x	x	x	
FM'_2C1	x	x	x						
FM'_2C2	x	x		x					
FM'_2C3	x	x	x		x	x			x
FM'_2C4	x	x		x	x	x			x

1. Les configurations partielles spécifiques. Lorsque l'extension (complète) d'un concept ne contient que des configurations valides d'un des deux modèles de départ, l'in-

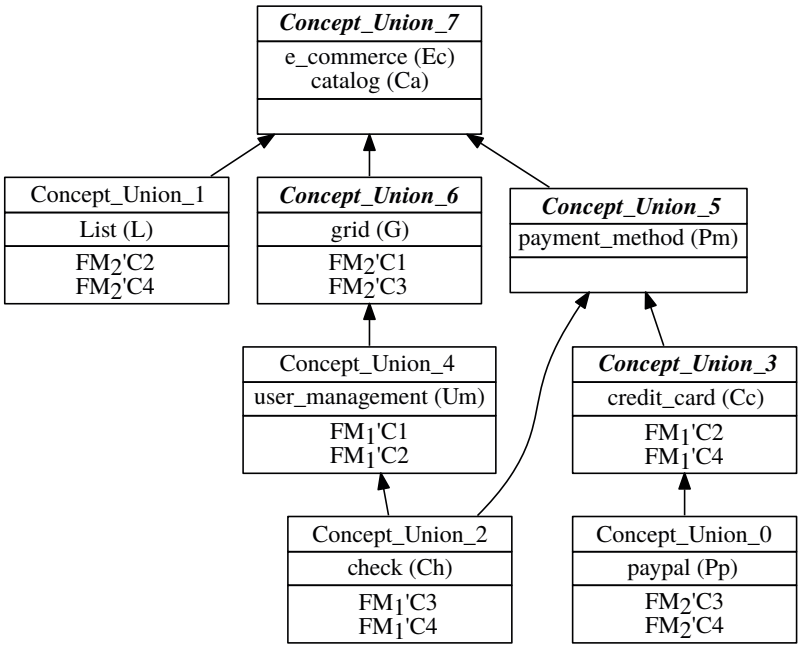


FIGURE 4.17 – AC-poset associé avec l'union stricte de FM'_1 et FM'_2.

tension est une configuration partielle ou bien une configuration de ce modèle. Dans les deux cas, cette combinaison de caractéristiques est spécifique à ce modèle et n'appartient pas à un cœur commun. Par exemple, dans l'AC-poset de la Figure 4.17, `Concept_union_0` représente une configuration partielle de FM'_2 et `Concept_union_2` une configuration partielle de FM'_1.

2. Les configurations partielles communes. Lorsque l'extension (complète) d'un concept contient des configurations des deux modèles, l'intension est une configuration partielle commune aux deux modèles. Plus spécifiquement :

- Si l'extension simplifiée contient au moins une configuration de chaque modèle, l'intension est une configuration valide des deux modèles. Ce cas n'apparaît pas dans l'exemple de la Figure 4.17. Cette configuration se retrouve alors dans leur intersection, qui n'est donc pas vide.
- Si l'extension simplifiée ne contient que des configurations d'un des deux modèles (comme pour le `Concept_Union_6` de la Figure 4.17, dont l'extension simplifiée ne contient que des configurations de FM'_2), l'intension est une configuration partielle stricte du modèle n'ayant pas de configuration dans l'extension simplifiée (ici FM'_1) et une configuration valide pour l'autre modèle (ici FM'_2).
- Si l'extension simplifiée est vide (comme pour `Concept_Union_7` et `Concept_Union_5` dans la Figure 4.17), l'intension est une configuration partielle stricte pour les deux modèles. C'est-à-dire qu'elle n'est valide pour aucun des deux modèles, mais qu'elle est contenue par des configurations valides des deux modèles : elle met donc en évidence des similitudes entre ces modèles.

Quand l'intersection est réduite ou bien vide, les concepts représentant les configurations partielles communes sont particulièrement utiles pour explorer plus en profondeur les traits communs aux deux modèles. Ils représentent des configurations possiblement incomplètes, desquelles des caractéristiques spécifiques présentes dans un seul modèle (telles que `Um` ou `Pp`), ou bien des combinaisons spécifiques de caractéristiques ont été enlevées. À partir de ces concepts, on peut construire une intersection approximative (denotée $\cap^{\sim}$). Pour

cela, on construit un contexte formel regroupant les configurations partielles communes extraites de l'AC-poset associé avec l'union stricte. Par exemple, de l'AC-poset associé avec l'union stricte de FM'_1 et FM'_2, on ne garde que les concepts `Concept_Union_3`, `Concept_Union_5`, `Concept_Union_6` et `Concept_Union_6`. L'intension de chaque concept représente alors un objet (une ligne) dans le contexte formel $FM'_1 \cap^{\sim} FM'_2$. On assigne à chaque configuration partielle commune le nom de son concept correspondant.

La Figure 4.18 montre le contexte formel (à gauche) et l'AC-poset (à droite) associés avec $FM_1 \cap^{\sim} FM_2$ et $FM'_1 \cap^{\sim} FM'_2$ (qui sont identiques). La Figure 4.19 (gauche) montre l'ECFD extrait de cet AC-poset. Il n'est pas approprié dans ce cas de construire les groupes et les mutex. Par exemple, dans les intensions des concepts représentant les configurations partielles communes, les caractéristiques `G` et `Pm` n'apparaissent jamais ensemble, alors qu'elles peuvent apparaître ensemble dans des configurations valides complètes. Les informations qui peuvent être identifiées grâce aux configurations partielles sont les co-occurrences (i.e., les relations obligatoires et les doubles implications transverses) et les implications binaires (i.e., les relations optionnelles et les implications transverses). Dans cet exemple, deux modèles possibles peuvent être dérivés (centre et droite de la Figure 4.19). Dans ce cas spécifique, on considère que le concepteur choisit le modèle ayant `e_commerce` comme racine.

	Ec	Ca	G	Pm	Cc
Conc._Union_7	x	x			
Conc._Union_6	x	x	x		
Conc._Union_5	x	x		x	
Conc._Union_3	x	x		x	x

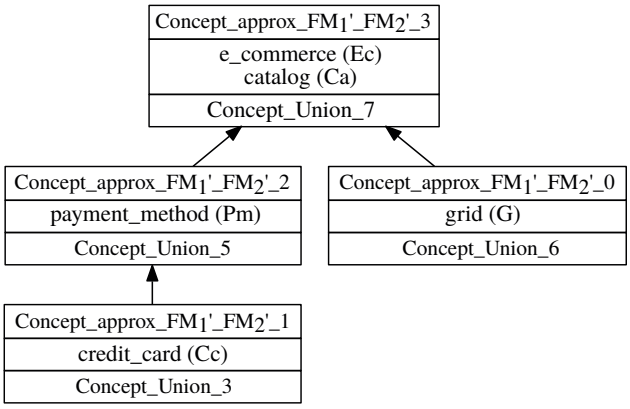


FIGURE 4.18 – (Gauche) : Le contexte formel $FM_1 \bigcap^{\sim} FM_2$ et $FM'_1 \bigcap^{\sim} FM'_2$, dont les objets sont les configurations partielles communes représentées par les concepts de l'AC-poset de la Figure 4.17 ; (Droite) : L'AC-poset associé à ce contexte formel

Ici, l'intersection approximative est simple, mais dans le cas général, des ECFDs complexes peuvent être trouvés et le concepteur peut pleinement bénéficier de leur potentiel. Comparée à une approche logique, la construction du modèle d'une intersection approximative est guidée grâce à l'ECFD et au mapping. Comparée à une approche basée sur la hiérarchie des modèles, aucune présupposition n'est faite sur les relations ontologiques : les structures conceptuelles et les ECFDs associés contiennent toutes les décisions pouvant être prises par le concepteur concernant la hiérarchie, qu'il peut modifier dans les limites du mapping.

4.3.4 Évaluation

Dans ce qui suit, nous cherchons à évaluer l'applicabilité de la méthode de composition proposée, ainsi que l'utilité de l'intersection approximative. Nous formulons donc les deux questions de recherche suivantes :

QR1 : Quel est l'aspect des ECFDs associés aux contextes-union et aux

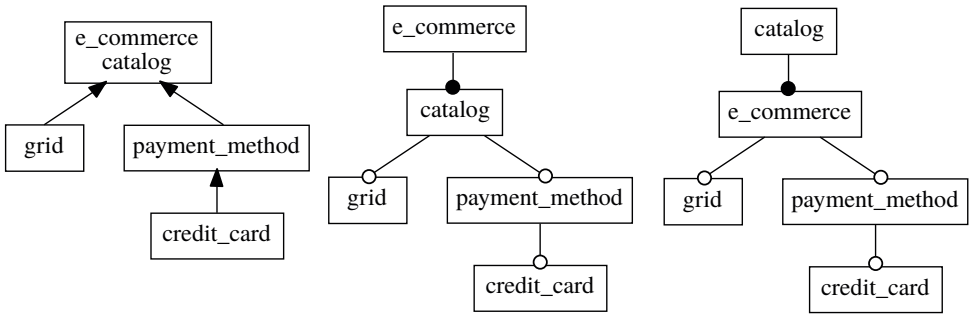


FIGURE 4.19 – (Gauche) ECFD de l'intersection approximative ; (Droite) Deux FMs pouvant être dérivés de cet ECFD

.

contextes-intersection ? Nous avons étudié dans la section précédente l'aspect des ECFDs construits à partir de FPLs de tailles diverses. Ici, nous cherchons à connaître leurs aspects lorsqu'ils sont issus de contextes-union et -intersection. Sont-ils similaires aux ECFDs précédents ? Leur taille est-elle raisonnable pour guider la synthèse de FMs ?

QR2 : Quel est l'apport de l'intersection approximative comparée à l'intersection traditionnelle ? Du fait de sa définition, l'intersection traditionnelle peut produire des modèles vides ou de très petite taille dans certains cas. Nous voulons ici étudier la pertinence de l'intersection approximative pour limiter l'obtention de modèles vides dans les cas cités ci-dessus.

Données et méthodologie

Pour répondre à ces deux questions, nous avons implémenté le processus présenté en Figure 4.20.

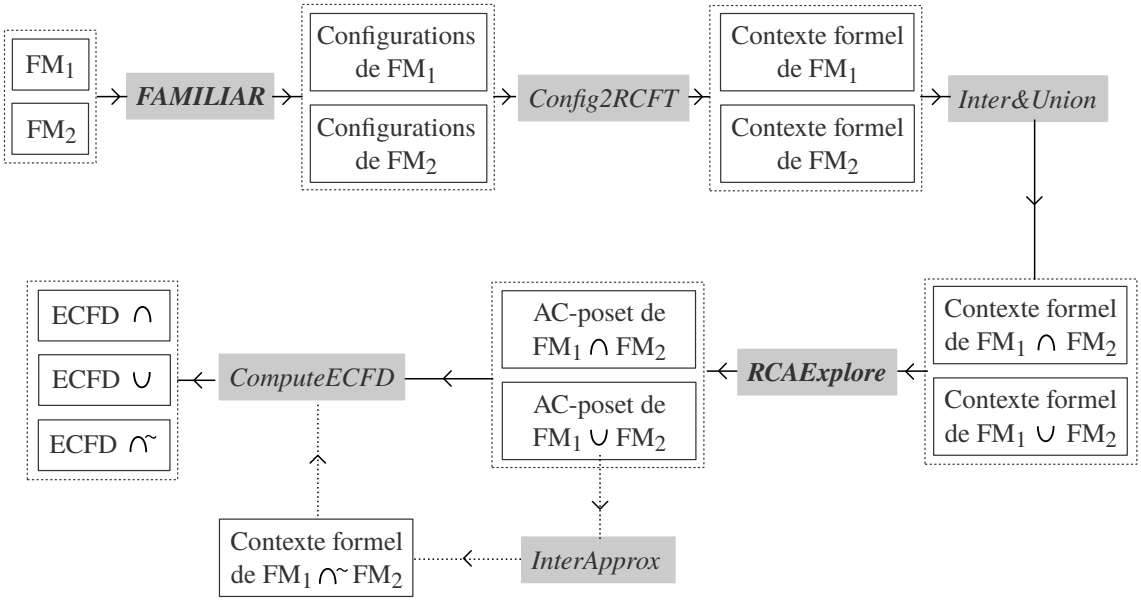


FIGURE 4.20 – Processus pour obtenir les ECFDs d'union, d'intersection et d'intersection approximative à partir de FMs

Ce processus utilise deux outils existants que nous avons déjà présentés dans les sections précédentes : `Familiar` [ACLF13] pour obtenir l'ensemble des configurations valides d'un

FM et `RCAExplore` pour construire l'AC-poset à partir d'un contexte formel. Ces outils sont représentés en gras et italique dans le processus. Nous avons aussi développé 4 programmes spécifiques pour réaliser les tâches suivantes :

- `Config2RCFT`, qui prend un ensemble de configurations calculé par `Familiar` et le transforme en un contexte formel, enregistré dans un fichier au format `.rcft` qui est lisible par `RCAExplore` ;
- `Inter&Union`, qui prend deux contextes formels au format `.rcft` et qui produit deux contextes représentant l'union et l'intersection ;
- `InterApprox`, qui prend l'AC-poset associé au contexte-union et calcule le contexte formel représentant l'intersection approximative ;
- `ComputeECFD`, qui calcule l'ECFD associè à un AC-poset.

Dans un premier temps, nous utilisons `Familiar` pour obtenir l'ensemble des configurations valides des deux FMs de départ, dont nous cherchons à réaliser une composition. Ces deux ensembles de configurations sont stockés dans deux fichiers textes différents, qui sont ensuite traités par `Config2RCFT` pour donner deux fichiers `.rcft`. Nous appliquons ensuite `Inter&Union` sur ces deux fichiers `.rcft` et nous obtenons deux nouveaux fichiers `.rcft`, représentant le contexte-union et le contexte-intersection des deux FMs de départ. Nous utilisons alors `RCAExplore` sur ces deux contextes pour construire les deux AC-posets associés. Avant de construire les ECFDs depuis ces AC-posets, nous calculons le contexte correspondant à l'intersection approximative depuis l'AC-poset associé au contexte-union avec `InterApprox`. Nous obtenons ainsi un troisième contexte formel, dont nous calculons l'AC-poset. Enfin, les ECFDs correspondant à ces 3 AC-posets sont calculés grâce au programme `ComputeECFD`.

Nous avons appliqué ce processus sur plusieurs FMs, possédant entre 4 et 864 configurations et entre 6 et 26 caractéristiques. Ils proviennent du répertoire de `SPLOT`[2] [MBC09], du site de `Familiar`[3] ou bien de la littérature. Nous avons aussi réalisé des variantes de ces FMs. Dans la Table 4.7, nous donnons le nom, le nombre de configurations (`#conf.`), de caractéristiques (`#car.`), de groupes (`#Xor` et `#Or`) et de contraintes transverses (`#Cst`) de chaque FM sélectionné.

TABLE 4.7 – FMs sélectionnés pour l'expérimentation

FMs	#car.	#conf.	#Xor	#Or	# Cst
FM1 (e-com.)	6	4	0	1	0
FM2 (e-com.)	6	4	1	0	0 4
Martini Eshop	11	8	1	1	1
Tang Eshop	10	13	1	1	2
Toacy Eshop	12	48	1	2	0
Wiki-V1	14	10	4	0	4
Wiki-V2 (V1 var.)	17	50	4	1	4
Wiki-V3 (V1 var.)	18	120	3	2	6
Bicycle1	19	64	2	0	2
Bicycle2	22	192	5	0	1
Bicycle3	25	576	4	0	2
Bicycle4	26	864	5	0	2

2. http ://www.splot-research.org/
3. http ://familiar.variability.io/

La Table 4.8 présente des informations sur l'union, l'intersection et l'intersection approximative. La première colonne (`FMs + op. compo.`) montre le nom des FMs et le type d'opération de composition. Les 2 colonnes `#car.` et `#conf. part.` présentent respectivement le nombre de caractéristiques dans le contexte formel associé à l'opération de composition et le nombre de configurations partielles (correspondant au nombre de concepts dans l'AC-poset).

QR1 : Afin d'avoir une idée de l'aspect des ECFDs de composition, nous avons rassemblé dans les 5 sous-colonnes `ECFD` de la Table 4.8 le nombre de groupes *xor*, de groupes *or*, de mutex, de cas de multi-parenté et de blocs des ECFDs obtenus grâce au processus présenté précédemment. Les cellules correspondant aux groupes *or* et *xor* et aux mutex sont remplies par NA ("non applicable") dans le cas de l'intersection approximative car, comme nous l'avons vu, ces informations ne sont pas cohérentes sur des configurations partielles.

QR2 : Pour évaluer l'utilité de la construction d'une intersection approximative pour éviter d'obtenir des modèles vides, nous avons comparé les intersections approximatives, en termes de nombre de configurations partielles (`#blocs`) et de caractéristiques (`#car`), par rapport aux deux autres opérations de composition. Pour réaliser cette comparaison, nous avons appliqué les trois opérations de composition sur les mêmes paires de FMs. La comparaison est exprimée en termes de pourcentage et est consignée dans les deux dernières colonnes de la Table 4.8

TABLE 4.8 – ECFDs d'intersection, d'intersection approximative et d'union stricte.

FMs + op. compo.	Contexte formel		ECFD					% $\neq \cap^{\sim}$	
	#car	#conf. part.	#Xor	#Or	#mutex	#multi par.	#blocs	#car	#blocs
FM1 ∩ FM2	5	2	0	0	0	0	2	-0%	-50%
FM1 $\cap^{\sim}$ FM2	5	4	NA	NA	NA	0	4	.	.
FM1 ∪ FM2	7	6	1	1	1	1	6	+40%	+50%
Martini∩Tang	0	0	0	0	0	0	0	-100%	-100%
Martini $\cap^{\sim}$ Tang	9	6	NA	NA	NA	0	6	.	.
Martini∪Tang	12	21	1	2	3	1	8	+33%	+33%
Martini∩Toacy	0	0	0	0	0	0	0	-100%	-100%
Martini $\cap^{\sim}$ Toacy	9	6	NA	NA	NA	0	6	.	.
Martini∪Toacy	14	56	1	1	4	0	10	+56%	+67%
Tang∩Toacy	8	5	1	2	0	0	5	-11.1%	-16.7%
Tang $\cap^{\sim}$ Toacy	9	6	NA	NA	NA	0	6	.	.
Tang∪Toacy	13	56	1	1	4	1	10	+44%	+67%
WikiV1∩WikiV2	0	0	0	0	0	0	0	-100%	-100%
WikiV1 $\cap^{\sim}$ WikiV2	11	7	NA	NA	NA	0	7	.	.
WikiV1∪WikiV2	20	60	5	6	26	0	16	+82%	+129%
WikiV1∩WikiV3	0	0	0	0	0	0	0	-100%	-100%
WikiV1 $\cap^{\sim}$ WikiV3	9	5	NA	NA	NA	0	5	.	.
WikiV1∪WikiV3	23	130	3	4	42	0	18	+156%	+260%
WikiV2∩WikiV3	11	6	2	0	1	0	6	-0%	-45.5%
WikiV2 $\cap^{\sim}$ WikiV3	14	11	NA	NA	NA	0	11	.	.
WikiV2∪WikiV3	21	164	6	14	10	1	17	+50%	+55%
Bicycle1∩Bicycle2	14	8	1	0	0	0	6	-6.7%	-40%
Bicycle1 $\cap^{\sim}$ Bicycle2	15	10	NA	NA	NA	0	10	.	.
Bicycle1∪Bicycle2	26	248	6	1	32	2	21	+73%	+110%
Bicycle3∩Bicycle4	23	288	5	1	8	0	18	-4.2%	-5.3%
Bicycle3 $\cap^{\sim}$ Bicycle4	24	19	NA	NA	NA	0	19	.	.
Bicycle3∪Bicycle4	27	1152	6	1	8	0	22	+13%	+16%

Analyse des résultats

QR1 : Les ECFDs obtenus avec les opérations de composition sont assez similaires à ceux que nous avions étudiés dans la section précédente. Là encore, on retrouve un petit nombre de groupes de caractéristiques : 16 groupes *or* et 6 groupes *xor* dans le pire des cas. On note cependant une augmentation du nombre de mutex, surtout dans le cas des opérations d'union. Cela s'explique par la définition du contexte-union : si une caractéristique est présente dans un contexte mais pas dans l'autre, on la garde tout de même dans le contexte-union. Les caractéristiques n'apparaissant jamais ensemble dans une configuration du fait de leur absence dans l'un des contextes de départ expliquent le nombre important de mutex dans ces ECFDs. Le nombre de cas de multi-parenté (6 pour 27) semble assez cohérent avec celui observé dans la section précédente qui était de 3 pour 10.

Nous pouvons observer la différence importante entre la taille (en termes de blocs) des ECFDs d'union et des ECFDs d'intersection. Ces derniers sont bien plus petits et très souvent inexistants. Sur les 9 paires de FMs étudiées, 4 donnent une intersection vide.

Les ECFDs construits ont une taille raisonnable comparée à celle des FMs de départ, avec un nombre de blocs allant de 2 à 22, de groupes allant de 0 à 16 et très peu de cas de multi-parenté. Ces résultats sont encourageants si nous considérons qu'un expert doit extraire un FM guidé par ces ECFDs.

QR2 : Les deux dernières colonnes de la Table 4.8 montrent la différence entre l'intersection et l'intersection approximative, l'union stricte et l'intersection approximative, et ce en termes de nombre de caractéristiques et de nombre de blocs (configurations partielles). Par exemple, si nous analysons FM1 $\cap^{\sim}$ FM2, nous pouvons observer que le nombre de caractéristiques de l'union FM1 $\cup$ FM2 est 40% plus importante que l'intersection approximative FM1 $\cap^{\sim}$ FM2. Le nombre de configurations partielles de FM1 $\cup$ FM2 est 50% plus élevé que celui de FM1 $\cap^{\sim}$ FM2. Lorsque l'intersection est importante, l'intersection approximative lui est bien souvent similaire. Au contraire, lorsque l'intersection n'est pas importante, voire inexistante, l'intersection approximative fournit des ECFDs de taille non négligeable.

Les fluctuations entre les tailles des ECFDs obtenus avec différentes opérations de composition peuvent donner des informations supplémentaires. Quand l'intersection est vide, une faible différence entre l'intersection approximative et l'union stricte (comme le cas Martini-Tang) indique une similarité non négligeable entre les FMs de départ, qui n'est pas mise en évidence par leurs sémantiques de configurations. À l'inverse, lorsque l'intersection est vide mais que la différence entre l'intersection approximative et l'union stricte est importante (comme le cas Bicycle3-Bicycle4), la similarité entre les deux FMs de départ n'est pas très importante. Lorsque l'intersection approximative est proche de l'intersection, cela signifie que la sémantique de configurations est bien représentée par les caractéristiques et les configurations partielles communes. Si le nombre de caractéristiques de l'intersection approximative est proche de celui de l'intersection (cas de Bicycle1-Bicycle2 avec -6.7%), mais que ce n'est pas le cas pour le nombre de blocs (-40%), alors cela signifie qu'il y a beaucoup de caractéristiques communes, mais que la sémantique de configurations n'est pas bien représentée par les configurations partielles communes. Ces informations peuvent aussi guider un expert durant le processus de composition de FMs.

En résumé (4.3) :

— La **composition des FMs** est une **opération d'analyse et de conception** essentielle à la bonne gestion d'une LPL.

— L'**union** et l'**intersection** ont une place spéciale dans la manipulation des modèles afin de donner des **vues différentes d'un système**.
— Traditionnellement, leur implémentation repose soit sur la **hiérarchie des modèles**, soit sur leurs **formules propositionnelles** associées.
— La méthode proposée ici repose sur la **sémantique de configuration** et **l'union/intersection ensembliste**, encodées sous la forme de contextes formels.
— Elle ne nécessite donc pas de FMs, ouvrant ainsi le champ d'applicabilité de ces opérations aux **familles de produits**.
— La **synthèse de modèle** définie dans la section précédente permet de construire des modèles associés aux contextes formels représentant l'union ou l'intersection d'ensembles de configurations.
— Grâce aux structures conceptuelles, nous définissons une **intersection approximative**, plus souple que l'intersection, qui met en évidence les **configurations partielles communes** à deux familles de produits.

4.4 Recherche exploratoire dans les variantes existantes

La sélection (ou configuration) de produits est une opération largement étudiée dans le domaine des LPLs et qui consiste à guider l'utilisateur dans le choix d'une configuration à travers la sélection et/ou la dé-sélection de caractéristiques [SBA+14, FFA+13, PRB11]. Comme la plupart des opérations des LPLs, la sélection de produits repose généralement sur la modélisation de la variabilité : la structure des FMs permet entre autres de dériver des configurateurs utilisés pour guider l'utilisateur dans son choix de configuration à travers des questions définies grâce aux contraintes du modèle.

Dériver un configurateur à partir d'une FPL n'ayant pas de FM soulève certains challenges. Les FMs n'étant pas logiquement complets [CW07], un modèle extrait automatiquement peut éventuellement proposer de sélectionner des configurations non valides. De plus, si sa sémantique ontologique n'est pas correcte, le configurateur dérivé souffrira d'incohérences [BABN16]. Somme toute, les stratégies de sélection de produits existantes sont difficiles à appliquer pour des FPLs d'ayant pas de modèles de variabilité corrects. Notons aussi que de tels configurateurs couvrent certains cas d'utilisation et pourraient bénéficier d'être complétés par d'autres stratégies de sélection de produits.

Dans cette section, nous passons dans un premier temps en revue les stratégies existantes de sélection de produits (Section 4.4.1). Nous présentons ensuite la recherche exploratoire et discutons de sa pertinence comme stratégie alternative et complémentaire de sélection de produits, basée sur les structures de l'AFC et non sur les FMs (Section 4.4.2) Nous proposons une implémentation de cette stratégie de sélection basée sur la génération locale dans les AOC-posets dans le but de faciliter le passage à l'échelle (Section 4.4.3) et nous testons notre méthode sur des jeux de données réels (Section 4.4.4).

4.4.1 La configuration de produits

Rabiser définit la *configuration de produits* comme étant "le processus de prendre des décisions pour sélectionner un produit particulier d'une LPL" [Rab09]. Dans la littérature, ce processus est très souvent impossible à distinguer de celui de *configuration de FMs*, qui consiste à "sélectionner et dé-sélectionner des caractéristiques d'un FM jusqu'à obtention d'une configuration valide" [BFGR13]. La configuration de produits depuis un modèle de

caractéristiques a été identifiée très tôt comme un défi du domaine des LPLs [PBvdL05] et a été longuement étudiée par la suite.

Plusieurs outils implémentent la configuration de FMs : nous illustrons ici les mécaniques de cette opération sur l'outil `SPLOT`[4] [MBC09], disponible en ligne gratuitement. `SPLOT` implémente le processus de *configuration interactive basée sur les caractéristiques* qui consiste à présenter un FM à un utilisateur, dans lequel il peut sélectionner et désélectionner des caractéristiques. De plus, ce configurateur propage automatiquement ces décisions de sorte qu'il ne soit pas possible d'y introduire de contradictions. Les caractéristiques sont présentées sous la forme d'un modèle dans lequel sont annotés les différents choix que peut faire l'utilisateur. La Figure 4.21 représente un configurateur dérivé du modèle sur les e-commerces.

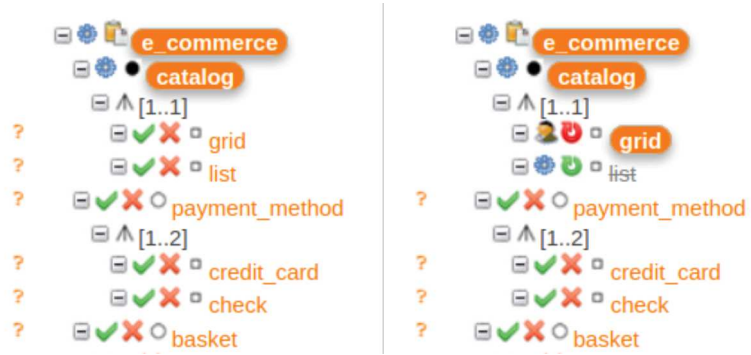


FIGURE 4.21 – (Gauche) Configurateur sans aucune décision de l'utilisateur. (Droite) Configurateur après que l'utilisateur ait sélectionné `grid`

À gauche est présenté le modèle sans décision de l'utilisateur. Automatiquement, les caractéristiques `e_commerce` et `catalog` sont sélectionnées car elles représentent les *core features* de la LPL. Le symbole d'engrenage présent à gauche des deux caractéristiques spécifie que ces choix ont été propagés automatiquement. À droite est présenté le même modèle après que l'utilisateur ait sélectionné `grid`. Une icône représentant une personne est présent à sa gauche indiquant que le choix vient de l'utilisateur. La caractéristique `list` est barrée du fait d'une propagation automatique de la décision de l'utilisateur : en effet, pour ne pas introduire de contradiction à cause de la cardinalité 1..1 du groupe *xor*, `list` ne doit pas pouvoir être sélectionnée ultérieurement. Les quatre points d'interrogation indiquent les décisions restantes. Lorsqu'il n'y a plus de points d'interrogation sur la gauche, le configurateur assure que la configuration est valide.

Plusieurs problèmes peuvent se poser sur des modèles plus grands que celui utilisé ici et des stratégies sont mises en place pour faciliter le processus de configuration dans ces cas-là.

Par exemple, dans le configurateur de `SPLOT`, l'utilisateur a la possibilité de changer de décision (en dé-sélectionnant une caractéristique sélectionnée, ou bien l'inverse), auquel cas le configurateur affichera les possibles contradictions que cela engendre. La Figure 4.22 montre la contradiction engendrée par la sélection forcée de `list`, i.e., `grid` et `list` ne peuvent être sélectionnées en même temps.

Lorsque les modèles présentent un grand nombre de caractéristiques, le nombre d'étapes à réaliser durant le processus de sélection est aussi potentiellement élevé. Tan et al. [TLYZ13] proposent une nouvelle approche ayant pour but d'identifier les points de variabilité (i.e., les décisions que peut prendre un utilisateur) qui possèdent le plus d'impact sur

4. http ://www.splot-research.org/

FIGURE 4.22 – Conflit engendré par la sélection forcée de `list`

la propagation automatique. De ce fait, en proposant à l'utilisateur de prendre ces décisions en premier, le nombre de décisions restant en est fortement réduit. Botterweck et al. [BTN+08] utilisent des techniques de visualisation pour faciliter le processus de configuration dans de grands modèles. On retrouve par exemple l'affichage horizontal et incrémental du modèle permettant de visualiser plus facilement un plus grand nombre de niveaux, ou encore la possibilité de se concentrer sur et d'agrandir une partie spécifique du modèle et de cacher le reste. Ils proposent aussi de visualiser les conséquences des décisions prises par l'utilisateur, c'est-à-dire de représenter graphiquement la sélection ou la dé-sélection automatiques de caractéristiques engendrées par une décision.

Benavides et al. [BTRC05] utilisent des problèmes de satisfaction de contraintes pour représenter les configurations d'un modèle en modélisant les contraintes inhérentes aux FMs. Il proposent à l'utilisateur d'appliquer des contraintes, appelées filtres, pour réduire l'espace des configurations valides et ainsi d'explorer un ensemble plus restreint de configurations correspondant à leurs exigences.

Enfin, Sannier et al. [SBA+14] discutent des similarités et de la complémentarité des approches basées sur les configurateurs et les comparateurs. Un comparateur prend généralement la forme d'une table décrivant un ensemble de produits en fonction d'un ensemble de caractéristiques. Leurs similarités proviennent principalement du fait que les deux approches représentent de manière différente la variabilité d'un ensemble de produits. Ils insistent sur l'apport d'un FM pour définir ces deux stratégies de configuration de produits ; un des défis identifiés dans cet article est la synthèse de FMs depuis ces tables de descriptions de produits.

Ces approches ont un point commun : leur implémentation nécessite un FM pour représenter la variabilité de l'ensemble des produits. Cela soulève une première limite considérable du fait de notre postulat de départ : **Comment guider un utilisateur dans le choix d'un produit d'une FPL si celle-ci ne possède pas de FM ?** Par la suite et afin d'être en accord avec notre étude des FPLs n'ayant pas de FM, nous donnons une définition de ce processus qui s'en désolidarise : la configuration de produits est une opération qui consiste à guider l'utilisateur vers une variante de la FPL à travers la sélection et la dé-sélection des caractéristiques qui distinguent ces variantes.

Dans la suite de cette section, nous étudions la recherche exploratoire afin de compléter les approches de sélection de produits existantes et d'appliquer cette opération sans l'aide d'un FM.

4.4.2 La recherche exploratoire pour la sélection de produits

La recherche exploratoire

La recherche d'information [SMR08] regroupe des activités visant à extraire de l'information correspondant à certains critères depuis un ensemble de données. La stratégie de recherche d'information la plus répandue consiste à récupérer des documents correspondant à une requête, une requête étant une conjonction de descripteurs de documents. Cette approche présume souvent que l'utilisateur connaît le document à récupérer et les mots-clés (i.e., descripteurs) qui le décrivent. Cependant, cette hypothèse n'est pas vérifiée dans toutes les situations car l'utilisateur ne sait pas toujours quel document convient le mieux à ses besoins, ou n'a que des connaissances partielles de ce qu'il cherche. C'est notamment le cas lorsque l'utilisateur n'est pas familier avec les documents existants, ou bien lorsque cette collection de documents est trop grande pour être connue entièrement. Il est alors avantageux pour l'utilisateur de **découvrir** l'espace de recherche, c'est-à-dire d'examiner et d'apprendre les informations relatives aux documents et à leurs descripteurs au fur et à mesure de ses recherches.

La **recherche exploratoire** est une stratégie de recherche d'information qui consiste à guider l'utilisateur dans son processus de recherche dans une collection de documents (i.e., un espace de recherche) afin de l'aider à sélectionner celui qui correspond le mieux à ses besoins [Mar06, WR09]. Elle assiste l'utilisateur dans la découverte et l'examen progressifs des documents disponibles, en lui permettant de **naviguer** dans l'espace de recherche en fonction des descripteurs qu'il sélectionne. C'est une approche interactive permettant de construire et de modifier une requête progressivement, plutôt que de requérir une combinaison prédéterminée de descripteurs pour récupérer un document spécifique. Ce processus exploratoire est donc particulièrement adapté aux situations dans lesquelles l'utilisateur ne connaît pas précisément le document qui l'intéresse ni ses descripteurs.

Définition 4.8 (Recherche exploratoire [Mar06, WR09])**.** *Stratégie de recherche aidant l'utilisateur à définir progressivement sa requête, tout en découvrant les documents lui correspondant : la requête est construite graduellement par sélection et/ou dé-sélection à la volée de descripteurs, guidée par des suggestions de modification, accompagnant ainsi l'utilisateur dans une exploration pas-à-pas de l'espace de recherche en fonction de ses choix.*

Implémentation

Les structures de graphe sont utilisées comme supports pour ces processus de navigation en 1) représentant et structurant l'espace de recherche et 2) permettant à un utilisateur de s'y déplacer. Un exemple de stratégie de représentation peut déterminer que chaque nœud du graphe représente un document et les descripteurs qui lui correspondent ; l'ensemble des descripteurs d'un nœud modélise alors la requête qui fait référence au document correspondant. Un autre exemple de stratégie de représentation peut définir un espace de recherche moins strict, où chaque nœud représente une requête formulable par l'utilisateur et qui ne correspond donc pas toujours à un document existant. Dans les deux cas, les arcs représentent les modifications à effectuer pour passer d'une requête à une autre. La recherche exploratoire guide alors l'utilisateur dans sa navigation d'un nœud à l'autre de la structure. À chaque étape, l'utilisateur est positionné sur un nœud du graphe et plusieurs choix de navigation sont présentés à lui (i.e., les arcs sortants). Le nœud sur lequel l'utilisateur est positionné correspond à l'état de la requête courante (i.e., les descripteurs sélectionnés) et un arc sortant correspond à une modification (élargissement/rétrécissement) possible de la requête courante.

Exemple. La Figure 4.23 montre un exemple d'espace de recherche modélisé par un

graphe selon la première stratégie de représentation. Il y a trois nœuds, représentant trois documents et leurs descripteurs : le *document 1* est décrit par les descripteurs d_1 et d_4 et le *document 2* par les descripteurs d_1, d_2 et d_3. L'arc labellisé par $\{+\{d_2, d_3\}, -\{d_4\}\}$ entre les nœuds *document 1* et *document 2* indique qu'il faut ajouter les descripteurs d_2 et d_3 et enlever le descripteur d_4 à la requête référençant le *document 1* pour obtenir la requête référençant le *document 2*.

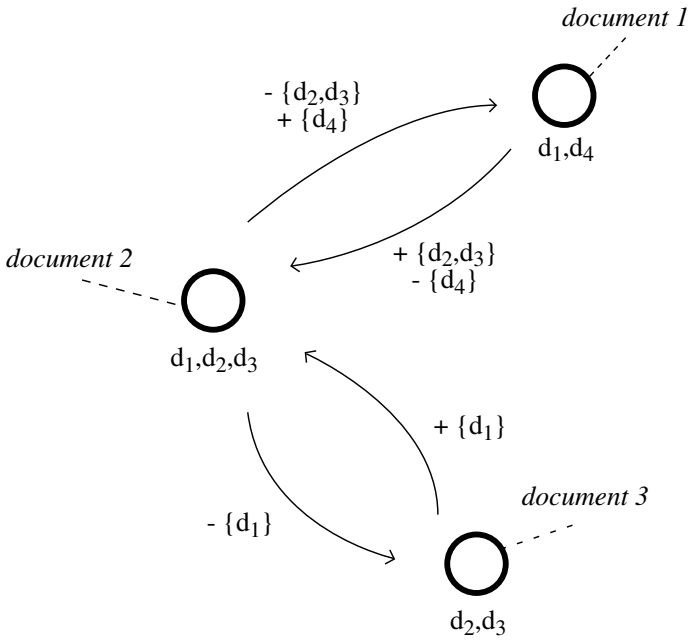


FIGURE 4.23 – Exemple de graphe représentant un espace de recherche composé de 3 documents et 4 descripteurs

Les structures de treillis ont été parmi les premières à être utilisées comme support à la recherche exploratoire [GSG86, GGP89] et leur usage dans ce domaine fut repris et complété plus tard par l'AFC [CN15]. Dans un treillis de concepts, l'état de la requête courante est défini par un concept : les descripteurs correspondent aux attributs de l'intension et les documents qu'ils référencent correspondent aux objets de l'extension. À chaque étape, l'utilisateur est placé sur exactement un concept (le concept courant) et les choix de navigation qu'il réalise en sélectionnant et/ou dé-sélectionnant des attributs le mènent à d'autres concepts : c'est la **navigation conceptuelle**. Le voisinage conceptuel d'un concept (i.e., ses super- et sous-concepts directs) représente les concepts les plus similaires à celui-ci et donc les modifications minimales de l'état de la requête courante : transiter vers un de ces concepts permet donc de naviguer dans l'espace de recherche par étapes minimales [GSG86] et en tombant dans des réponses non vides [Fer14]. Naviguer vers un sous-concept revient à sélectionner des attributs/descripteurs en plus et mène donc à une requête plus spécialisée. Naviguer vers un super-concept conduit à la dé-sélection d'attributs/de descripteurs et donc à une requête plus généralisée. La navigation conceptuelle par étapes non-minimales est aussi possible, en sélectionnant ou dé-sélectionnant arbitrairement des attributs et peut potentiellement conduire l'utilisateur à un concept éloigné du concept courant.

Exemple. La Figure 4.24 montre une partie du treillis de concepts présenté en Figure 4.25. Elle montre deux étapes de navigation conceptuelle. Le concept en gris est le concept courant. Dans cette structure, les concepts ne sont pas présentés de manière optimisée (i.e., les configurations et les caractéristiques sont présentes plusieurs fois). En gras sont représentés les éléments introduits dans le concept correspondant. Sont annotés

en orange les choix présentés à l'utilisateur. On peut voir sur la partie de gauche que le concept courant (`Concept_10`) introduit la configuration C_1 et la caractéristique `Strong`. Se déplacer vers son super-concept revient à dé-sélectionner `Strong` et se déplacer vers son sous-concept revient à sélectionner `Wireless`. Si l'on fait ce dernier choix, le concept courant devient le `Concept_7`, qui est neutre. De celui-ci, on peut faire le chemin inverse en dé-sélectionnant `Wireless`, ou bien continuer à sélectionner des caractéristiques (`Infrared` ou `Bluetooth`) et se déplacer vers ses sous-concepts.

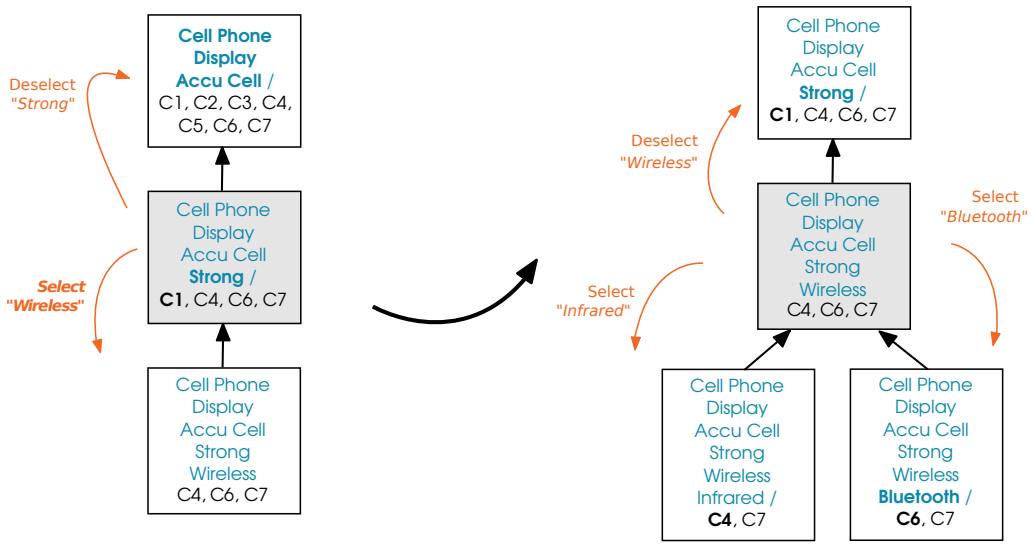


FIGURE 4.24 – Exemple de navigation conceptuelle par étapes minimales dans les treillis de concepts

La navigation conceptuelle est utilisée dans plusieurs applications, par exemple pour récupérer des documents web [CR04, DE07], naviguer dans une collection d'images [DVE06], ou encore explorer des ensembles de données bio-informatiques [SOM+05].

La recherche exploratoire pour la configuration de produits

Nous proposons d'utiliser la recherche exploratoire par navigation conceptuelle comme un moyen de réaliser de la sélection de produits dans les FPLs n'ayant pas de modèles de variabilité. Par la suite, on considère que les documents sont des configurations valides et que les descripteurs sont des caractéristiques. Nous avons vu que les configurateurs permettent de guider l'utilisateur grâce à un nombre réduit de questions vers une configuration qui répond aux décisions qu'il exprime. Ces questions, dérivées d'un modèle cohérent, respectent sa sémantique ontologique et sont donc cohérentes vis-à-vis du domaine. De plus, un configurateur peut inférer certaines décisions en fonction des contraintes du modèle : il respecte ainsi sa sémantique logique et ne permet à l'utilisateur que de choisir des configurations valides. Bien que cette approche présente de nombreux avantages, nous discutons ici des raisons qui ont motivé notre approche.

Les configurateurs reposent sur l'hypothèse que la FPL est décrite par un modèle de variabilité. Si celle-ci n'en a pas, ce modèle peut être généré manuellement ou automatiquement. Cependant, nous avons vu que la synthèse automatique pouvait générer des modèles incohérents, ou ayant un ensemble de configurations plus vaste que celui de départ. Si le modèle est incohérent, le configurateur dérivé le sera aussi. Si le modèle possède une sémantique de configurations plus large, l'utilisateur pourra sélectionner des produits logiciels non existants. Le modèle peut être complété par une formule propositionnelle

plus complexe que les contraintes transverses traditionnelles afin d'obtenir un ensemble de configurations équivalent, mais dans ce cas le configurateur pourrait inférer des décisions incompréhensibles par un utilisateur. La recherche exploratoire par navigation conceptuelle étant basée directement sur la description de la FPL, ces deux problématiques ne se posent pas. En effet, les structures conceptuelles incluent l'ensemble exact des configurations de départ. De plus, nous avons établi dans la Section 3.3 que les intensions des concepts d'un treillis correspondaient à toutes les configurations partielles de la famille de logiciels : à aucun moment de la navigation l'utilisateur ne formule de configurations invalides [Fer14]. Cependant, la méthode de sélection est moins intuitive du fait de l'absence de sémantique ontologique.

De manière générale, la recherche exploratoire présente un nouvel angle d'approche de la sélection et vient compléter les méthodes existantes en offrant un moyen de sélection plus flexible. En effet, les configurateurs ont tendance à présenter les questions à un utilisateur en fonction de ses choix précédents. Même si la configuration qu'il obtient à la fin de la sélection est valide et correspond à ses décisions, certaines questions ont pu être omises en fonction d'un choix réalisé tôt dans le processus de sélection. C'est-à-dire que l'utilisateur peut être amené à réaliser un choix sans connaître son impact sur le reste de la sélection. Dans le cas où l'utilisateur ne connaît pas précisément la configuration qu'il cherche, il doit tenter d'autres combinaisons qui pourraient aussi correspondre à ses attentes et qui auraient été mises hors d'atteinte lors des tentatives précédentes. Le fait est que, une fois une configuration obtenue grâce à un configurateur, celui-ci ne donne pas d'informations sur les configurations semblables. Démarrer un processus de recherche exploratoire à partir d'une configuration obtenue avec un configurateur permettrait à l'utilisateur d'explorer les autres configurations valides proches de la sienne sans redémarrer le processus. Réaliser l'intégralité de la sélection de produits par navigation conceptuelle est cependant plus long que de répondre aux questions d'un configurateur.

Avec la navigation conceptuelle, un utilisateur peut démarrer la sélection depuis :

— Une configuration existante, correspondant à un concept introducteur de configuration. Par exemple, le `Concept_4` de la Figure 4.25 (gauche), qui correspond à la configuration valide c_6.
— Une configuration partielle, correspondant à un concept (qui n'est pas le *bottom-concept*). Par exemple le `Concept_9` représentant la configuration partielle {`Cell Phone, Display, Accu Cell, Wireless`}.
— La plus petite configuration partielle possible, correspondant au *top-concept* et regroupant les caractéristiques cœur de la famille de logiciels. Dans la Figure 4.25, cette configuration partielle est {`Cell Phone, Display, Accu Cell`}.

Le type du concept courant donne aussi des informations à l'utilisateur : si c'est un introducteur de configurations, alors la configuration partielle courante est valide ; si c'est un introducteur de caractéristiques, alors l'ensemble des configurations dans l'extension de ce concept représente toutes les configurations ayant la caractéristique introduite. Les concepts neutres représentent des configurations partielles qui sont communes à plusieurs configurations valides. Prenons l'exemple de la Figure 4.25. Imaginons que l'utilisateur débute du `Concept_5`, représentant la configuration c_4 = {`Cell Phone, Display, Accu Cell, Strong, Wireless, Infrared`}. Trois choix s'offrent à lui. Il peut enlever soit `Infrared` en se déplacant vers le super-concept neutre `Concept_7`, soit `Strong` en se déplacant vers le super-concept `Concept_8`. Le `Concept_8` représente la configuration partielle {`Cell Phone, Display, Accu Cell, Wireless, Infrared`}, commune à toutes les configurations ayant la caractéristique `Infrared`. Enfin, il peut ajouter `Bluetooth` en se déplacant vers le sous-concept `Concept_1`, correspondant à la configuration valide c_7.

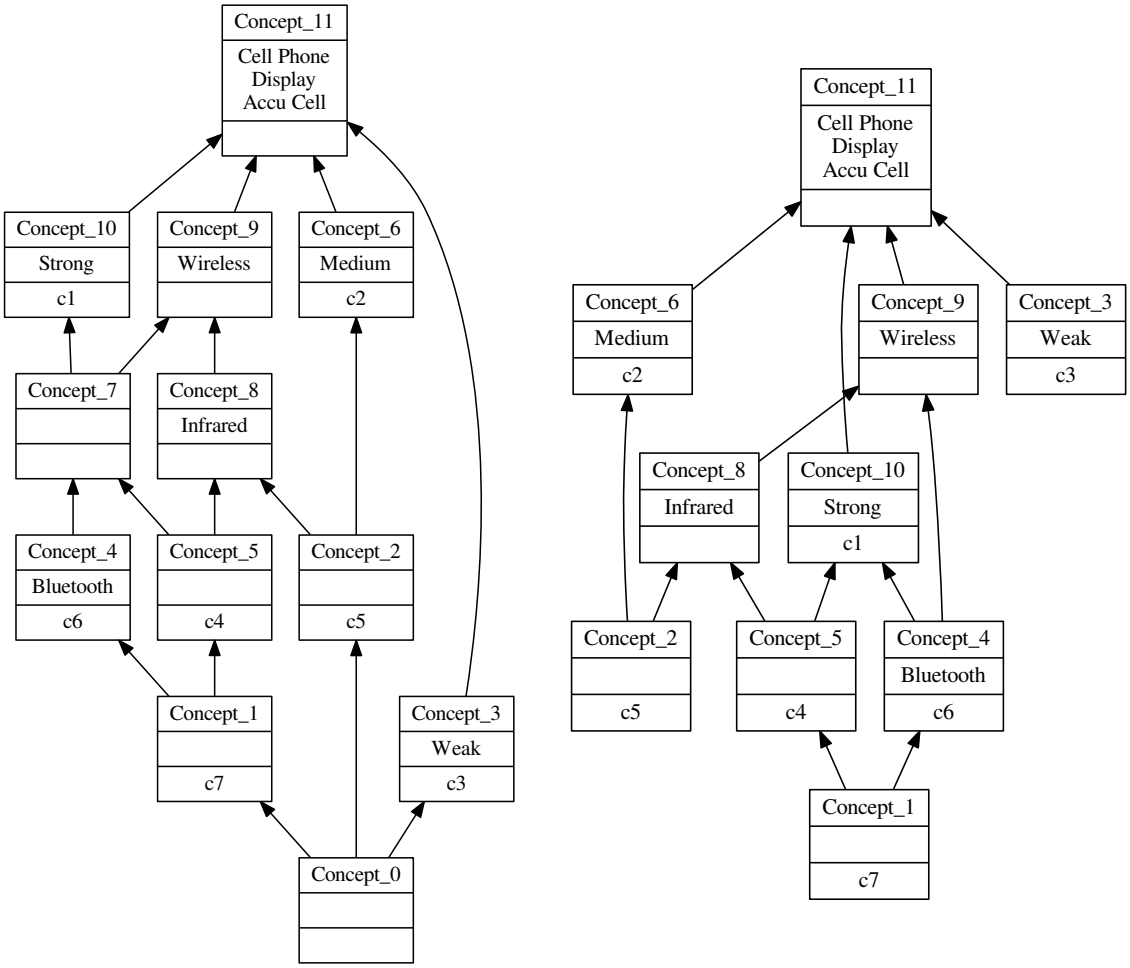


FIGURE 4.25 – Treillis des concepts (gauche) et AOC-poset (droite) associés à la FPL sur les e-commerces

4.4.3 Réduire la complexité de la recherche exploratoire par navigation conceptuelle

L'utilisation de l'AFC pour la recherche exploratoire soulève des problématiques [GGP89, CR04], notamment à cause de la taille (en termes de nombre de concepts et de la relation d'ordre) des treillis utilisés, connue pour grandir exponentiellement avec la taille des données d'entrée. Générer le treillis peut prendre du temps et nécessite des algorithmes adaptés pour être utilisé efficacement dans des applications. De plus, un utilisateur peut être désorienté s'il navigue par étapes minimales dans ce type de structures pouvant être larges et alambiquées. Dans ce qui suit, nous cherchons un moyen efficient d'appliquer la navigation conceptuelle.

État de l'art

Plusieurs méthodes ont été proposées dans la littérature pour réduire la complexité de la navigation conceptuelle. Dans [GGP89], Godin et al. choisissent de ne montrer qu'une partie du treillis à l'utilisateur, réduite au concept courant et à son voisinage, afin de simplifier la visualisation. Melo et al. [MGA13], proposent deux méthodes pour extraire des structures d'arbre depuis des treillis de concepts afin de visualiser et de naviguer dans des structures moins complexes. Carpineto et Romano [CR04] permettent d'appliquer dynamiquement des contraintes durant le processus de navigation et ainsi élaguer le treillis.

Cela permet de réduire l'espace de recherche explorable et aide l'utilisateur à se concentrer sur les parties qui l'intéressent. Dans la même ligne d'idées, Stumme et al. [STB+02] présentent les treillis *icebergs* qui sont des structures élaguées qui ne présentent que leur partie supérieure. Ces concepts correspondent aux concepts les plus généraux, i.e., ayant le moins d'attributs et correspondant à plus d'objets. Dans [DDE08], Dau et al. présentent `SearchSleuth`, un outil pour l'analyse locale de requêtes web basé sur l'AFC, qui dérive un concept et son voisinage depuis une requête donnée. Le domaine ne pouvant être calculé intégralement, cet outil génère un nouveau contexte formel à chaque étape de la navigation : pour chaque requête de l'utilisateur, une liste de résultats est d'abord récupérée, puis un ensemble de termes importants sont extraits et enfin un contexte est construit à partir des termes extraits et des documents correspondants. La navigation est gérée à travers une interface qui suggère des termes à l'utilisateur, rendant la structure de graphe sous-jacente ainsi que sa complexité implicites. Alam et al. [ALN16] présentent `LatViz`, un outil qui fournit plusieurs opérations pour réduire l'espace de recherche, facilitant ainsi la visualisation et la navigation. Les auteurs proposent par exemple d'afficher le treillis niveau par niveau : sélectionner un concept du niveau n affiche tous ses sous-concepts aux niveau $n-1$. Une autre fonctionnalité rend possible l'élagage du treillis de concepts à la manière de Carpineto et Romano [CR04]. De plus, l'outil permet de calculer l'AOC-poset comme support à la navigation conceptuelle : les auteurs décrivent l'AOC-poset comme le cœur du treillis. Greene et Fischer [GF16] discutent des approches d'élargissement et de rétrécissement qui ne sont pas restreintes au voisinage conceptuel afin de faciliter la navigation par étapes non-minimales dans de larges espaces de recherche

En résumé, les méthodes existantes proposent soit de faciliter la visualisation, soit de couper des parties du treillis. Dans ce qui suit, nous nous intéressons de plus près à ce que Alam et al. appellent le "cœur du treillis" [ALN16], c'est-à-dire l'AOC-poset, une sous-hiérarchie du treillis qui conserve les informations relatives à la variabilité.

L'AOC-poset pour la navigation conceptuelle

L'AOC-poset peut être utilisé comme une alternative plus réduite au treillis de concepts en ce qui concerne 1) la structuration d'un ensemble de configurations en fonction des caractéristiques qu'elles partagent et 2) la navigation dans cette collection par sélection et dé-sélection de caractéristiques. L'AOC-poset n'est pas un treillis élagué, mais bien une contraction de cette structure, car elle présente une sous-hiérarchie restreinte à certains concepts du treillis.

Alors que le treillis des concepts représente toutes les configurations partielles que peut formuler un utilisateur (voir Section 3.3), l'AOC-poset restreint ces configurations partielles à 1) l'ensemble des configurations valides (i.e., concepts introducteurs de configurations) et 2) les plus grandes configurations partielles contenant une caractéristique donnée (i.e., concepts introducteurs de caractéristiques). Les AOC-posets ne préservent pas la propriété de navigation par étapes minimales, car les concepts neutres ne sont pas pris en compte : les modifications possibles des requêtes sont alors naturellement factorisées afin de ne garder que les plus significatives. Ainsi, lorsque l'on ajoute une caractéristique, l'état de la requête courante transite soit vers une configuration valide plus spécifique, soit vers la configuration partielle maximale contenant cette caractéristique (i.e., la configuration partielle partagée par toutes les configurations ayant cette caractéristique).

Exemple. Considérons le treillis de concepts de la Figure 4.25 (gauche). Imaginons que le `Concept_4` soit le concept courant et que l'on dé-sélectionne la caractéristique `Bluetooth`, nous amenant ainsi sur le `Concept_7`. Depuis ce dernier, on peut dé-sélectionner soit `Strong` et atteindre le `Concept_9`, soit `Wireless` et atteindre le `Concept_10`. Dans l'AOC-poset de

la Figure 4.25 (droite), ces choix sont "condensés" du fait de l'absence de concepts neutres, jouant jusqu'ici le rôle "d'étapes de transition". Cette fois-ci, depuis le `Concept_4`, on peut soit dé-sélectionner `Bluetooth` et `Strong` en une seule étape et atteindre le `Concept_9`, soit dé-sélectionner `Bluetooth` et `Wireless` et atteindre le `Concept_10`.

L'AOC-poset peut donc être considéré comme une alternative plus réduite et condensée au treillis de concepts (qui peut être reconstruit à partir de l'AOC-poset). Cette sous-hiérarchie du treillis est naturellement plus petite : alors que la taille du treillis grandit de manière exponentielle, celle de l'AOC-poset est linéaire en la taille des données d'entrée. Malgré cela, elle peut cependant rester très grande. Explorer le voisinage conceptuel d'un concept ne nécessite pas de construire l'intégralité de la structure conceptuelle : nous explorons donc la piste de la génération locale dans l'AOC-poset.

Génération locale dans l'AOC-poset

La génération locale est une stratégie de construction qui consiste à ne générer qu'une partie de la structure. Cette stratégie a déjà été appliquée aux treillis de concepts, avec par exemple des algorithmes tels que *nextClosure* [Gan10] ou *iceberg* [STB$^+$02], permettant de construire les concepts pas-à-pas suivant un ordre lectique. À notre connaissance, 4 algorithmes existent pour la construction de l'AOC-poset : Ares [DDHL95], Ceres [Leb00], Pluton [BHM$^+$05] et Hermes [BGH$^+$14]. Cependant, aucun d'entre eux ne réalise de génération locale.

Dans ce qui suit, nous présentons un algorithme de génération locale du voisinage conceptuel d'un concept C_i dans l'AOC-poset. L'Algorithme 6 présente la manière de calculer la couverture supérieure. Son principe est le suivant : nous calculons d'abord les super-concepts de C_i qui sont des introducteurs d'attributs minimaux, puis les introducteurs d'objets entre C_i et les introducteurs d'attributs minimaux et enfin, nous éliminons les introducteurs d'attributs qui sont des super-concepts des introducteurs d'objets obtenus.

Nous illustrons les 5 étapes de l'algorithme sur le `Concept_1` de l'AOC-poset de la Figure 4.25 (gauche). Chaque étape est représentée sur l'AOC-poset correspondant dans la Figure 4.26 et est identifiée par son numéro. Notons que nous construisons ici la couverture supérieure dans l'AOC-poset sans avoir préalablement construit la structure : ce que nous montrons dans la Figure 4.26 est à titre indicatif.

❶ La couverture supérieure dans l'AOC-poset correspond aux plus petits concepts introducteurs d'attributs ou d'objets qui sont des super-concepts de $C_i = (X_i, Y_i)$. L'ensemble des super-concepts introducteurs d'attributs de C_i correspond aux concepts introduisant les attributs de son intension Y_i, car C_i hérite de tous les attributs introduits dans ses super-concepts. La Ligne 1 de l'Algorithme 6 récupère donc l'ensemble des attributs de l'intension de C_i dans la variable A_c ; leurs concepts introducteurs peuvent être obtenus en calculant les concepts $(\beta(a), \alpha \circ \beta(a))$ pour chaque attribut $a \in A_c$, mais ce calcul est repoussé à plus tard.

Exemple. Imaginons que nous souhaitions connaître les configurations similaires à la configuration c_7, composée des 7 caractéristiques suivantes :{`Cell Phone, Display, Accu Cell, Strong, Wireless, Infrared, Bluetooth`}. Le concept formel introduisant cette configuration est le `Concept_1` de la Figure 4.25 et nous cherchons donc son voisinage conceptuel. Les 7 caractéristiques de son intension sont placées dans A_c.

❷ Nous cherchons ensuite à identifier parmi ces attributs ceux qui sont introduits dans les plus petits super-concepts de C_i. Notons que, pour deux attributs a_1 et a_2, $(\beta(a_1), \alpha \circ \beta(a_1)) \geq (\beta(a_2), \alpha \circ \beta(a_2))$ si et seulement si $a_1 \in \alpha \circ \beta(a_2)$. De ce fait, les plus petits concepts introducteurs d'attributs peuvent être calculés à partir des attributs

Algorithme 6 : Couverture supérieure d'un concept dans l'AOC-poset

Input : Un concept $C_i = (X_i, Y_i)$
Output : La couverture supérieure de C_i dans l'AOC-poset

1 $A_c \leftarrow Y_i$
2 **foreach** $a \in Ac$ **do**
3 $\quad F_a \leftarrow \alpha \circ \beta(a)$
4 $\quad A_c \leftarrow A_c \setminus \{F_a \setminus \{a\}\}$
5 $AC_m \leftarrow \{(\beta(a), \alpha \circ \beta(a)) | a \in A_c\}$
6 $O \leftarrow \emptyset$
7 **forall** $C_a = (X_a, Y_a) \in AC_m$ **do**
8 $\quad O \leftarrow O \cup (X_a \setminus X_i)$
9 **forall** $o \in O$ **do**
10 $\quad$ **if** $\alpha(o) \not\subset Y_i$ **then**
11 $\quad\quad O \leftarrow O \setminus \{o\}$
12 **forall** $o \in O$ **do**
13 $\quad T = \{C = (X, Y) | (C \in AC_m) \wedge (o \in X)\}$
14 $\quad AC_m \leftarrow AC_m \setminus T$
15 $\quad Y_o \leftarrow \beta \circ \alpha(o)$
16 $\quad$ **if** $\exists p \in O, p \neq o$ *tel que* $p \in Y_o$ **then**
17 $\quad\quad O \leftarrow O \setminus \{o\}$
18 **return** $AC_m \cup \{(\beta \circ \alpha(o), \alpha(o)) |\ o \in O\}$

qui n'apparaissent pas dans la fermeture $\alpha \circ \beta$ d'au moins un des autres attributs de l'intension. Les Lignes 2 à 4 de l'Algorithme 6 calculent la fermeture F_a de chaque attribut de A_c et soustraient F_a privée de a de l'ensemble des attributs A_c. À la fin de la Ligne 4, les attributs restant dans A_c sont ceux introduits dans les introducteurs d'attributs minimaux : ces concepts sont calculés en Ligne 5 et conservées dans la variable AC_m.

Exemple. Prenons par exemple $F_{Infrared}$, qui est égal à {`Cell Phone, Display, Accu Cell, Wireless, Infrared`}. On retire `Infrared` de cette liste et on soustrait cette liste à A_c, qui est à présent égal à {`Strong, Infrared, Bluetooth`}. Ensuite, $F_{Strong} \setminus$ `Strong` = {`Cell Phone, Display, Accu Cell`}, qui sont des caractéristiques qui ne sont déjà plus dans A_c : A_c reste donc ici inchangé. Enfin, $F_{Bluetooth} \setminus$ `Bluetooth` ={`Cell Phone, Display, Accu Cell, Strong, Wireless`} : la caractéristique `Strong` est donc enlevée de A_c, qui est finalement réduit à {`Infrared, Bluetooth`} (i.e., AC_m).

❸ Chaque super-concept de C_i qui est un introducteur d'objet minimal est soit un super-concept, soit un sous-concept d'un introducteur d'attribut minimal. Puisque nous voulons les super-concepts introducteurs directs de C_i, les introducteurs d'objets minimaux que nous cherchons sont introduits entre C_i et les introducteurs d'attributs minimaux. Cela signifie aussi que les introducteurs d'attributs minimaux ayant un introducteur d'objet entre eux et C_i ne font pas partie de la couverture supérieure. Les Lignes 6 à 8 identifient les objets introduits entre C_i et les concepts introducteurs d'attributs minimaux et les conservent dans O. Pour chaque introducteur d'attribut minimal, on retire de son extension les objets de l'extension de C_i : en effet, les objets étant hérités des sous-concepts, les objets que nous cherchons sont dans l'extension d'un introducteur d'attributs minimal mais pas dans celle de C_i.

Exemple. β(`Infrared`) = $\{c_4, c_5, c_7\}$ et β(`Bluetooth`) = $\{c_6, c_7\}$. Lorsque l'on enlève l'extension du `Concept_1` (i.e., c_7), on obtient $O = \{c_4, c_5, c_6\}$.

❹ Cependant, il se peut qu'il y ait dans O des objets qui soient introduits dans des sous-concepts des introducteurs d'attributs minimaux, mais qui ne soient pas comparables avec C_i. On élimine ces cas-là dans les Lignes 9 à 11 : pour chaque objet de O, on calcule l'intension de son concept introducteur et on vérifie qu'elle soit incluse dans Y_i. Si elle ne l'est pas, l'objet est retiré de O.

Exemple. C'est le cas de la configuration c_5 : son intension, contenant entre autres la caractéristique `Medium`, n'est pas incluse dans celle du `Concept_1`. L'ensemble O est donc réduit à $\{c_4, c_6\}$.

❺ Enfin, on élimine les introducteurs d'attributs minimaux ayant un introducteur d'objet entre eux et C_i, car ils ne font pas partie de la couverture supérieure. Pour cela, la Ligne 13 sélectionne les introducteurs d'attributs de AC_m ayant au moins un objet de O dans leur extension et les retire de AC_m (Ligne 14). La Ligne 15 calcule l'extension de chaque introducteur d'objet et la Ligne 16 vérifie qu'il n'y ait pas d'autre objet de O dans cette extension. Si c'est le cas, le concept introduisant cet objet n'est pas minimal (ou bien est le même si $\beta \circ \alpha(o) = \beta \circ \alpha(p)$) car il existe au moins un autre introducteur d'objet entre lui et C_i : il est retiré de O (Ligne 17). Les objets restant dans O sont introduits dans des super-concepts minimaux de C_i : on retourne ces derniers concepts ainsi que les introducteurs d'attributs qui n'ont pas été retirés de AC_m (Ligne 18).

Exemple. L'extension du concept introduisant `Infrared` contenant un objet de O (i.e., c_4) est retiré de la couverture supérieure. Il en est de même du concept introduisant `Bluetooth`, sauf que celui ci n'est pas réellement retiré de la couverture supérieure, car il correspond au concept introduisant c_6 qui a été conservé dans AC_m.

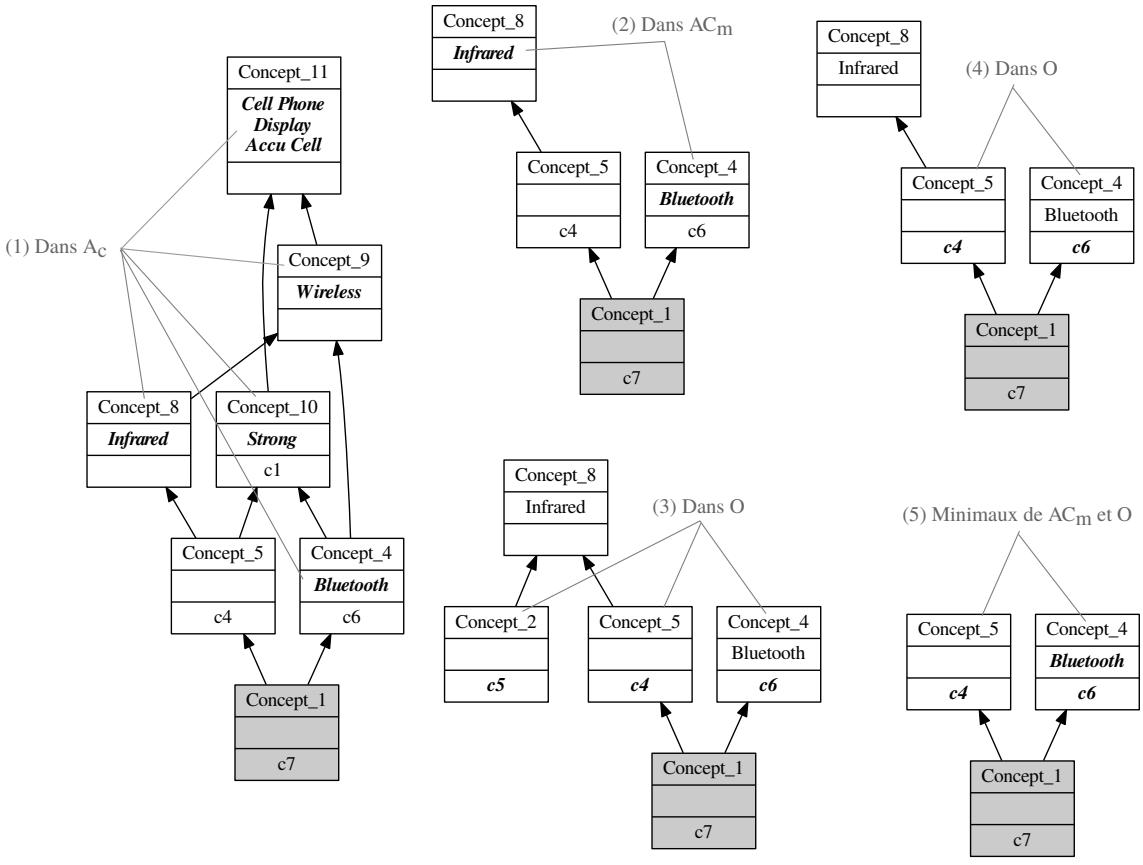


FIGURE 4.26 – Étapes de génération de la couverture supérieure dans l'AOC-poset du concept introduisant la configuration c_7

Le calcul de la couverture inférieure est le même que celui présenté dans l'Algorithme 6, sauf que les rôles des attributs et des objets sont échangés, du fait de la dualité des structures conceptuelles.

4.4.4 Évaluation

Nous avons cherché à évaluer quantitativement la méthode proposée à travers deux questions de recherche :

QR1 : La méthode proposée est-elle applicable sur des collections de données existantes ? En effet, la génération du treillis ou de l'AOC-poset en entier peut demander un temps de calcul assez long et donc potentiellement trop long pour être utilisé dans une application. À travers cette question, nous cherchons à déterminer si la génération locale peut se réaliser assez rapidement pour être utilisée par des applications à destination d'utilisateurs.

QR2 : Le gain de la génération locale d'AOC-poset est-il assez important pour justifier son utilisation ? Cette fois, nous voulons déterminer à quel point la génération locale dans l'AOC-poset réduit l'espace de recherche généré. Il est intéressant de réaliser cette comparaison en fonction du treillis de concepts, mais aussi en fonction de l'AOC-poset.

Données et méthodologie

Nous avons cherché des données qui correspondent à ces trois critères qui nous semblaient importants pour réaliser une évaluation pertinente :

— être issues de familles de logiciels réelles ;
— être de taille assez grande pour être significatives ;
— pouvant réellement bénéficier de la méthode proposée.

Nous avons choisi d'étudier des informations sur les différents *forks* de projets Github [5] (pouvant être vus comme des variantes d'un projet obtenues par *clone-and-own*), grâce à un outil académique nommé `fork-insight` [6] [RZK18]. Cet outil permet de sélectionner un projet de Github et de rassembler sous forme tabulaire les informations concernant les *forks* de ce projet. Les informations proposées incluent : le nombre de *commit*, le nombre de lignes de code modifiées, les dates de création et de dernière mise à jour, ainsi qu'un ensemble de mots clés représentatifs de la variante du projet source. Un des buts de `fork-insight` est de renseigner facilement l'utilisateur sur les propriétés des variantes déjà existantes d'un projet, pour qu'il sache si 1) la variante qu'il cherche existe déjà, ou si 2) il existe une variante proche de ce qu'il souhaite développer afin de modifier cette variante du projet plutôt que le projet source. `Fork-insight` permet de réaliser une recherche par mot-clés, afin de réduire le nombre de variantes affichées dans la table présentée à l'utilisateur. Trouver une variante existante ou une variante proche revient à faire de la sélection de produits et la recherche exploratoire est particulièrement adaptée dans ce contexte. Nous pensons que générer le voisinage conceptuel pour une variante donnée, ou bien pour un ensemble de mots clés donné peut compléter les fonctionnalités fournies par `fork-insight` pour aider l'utilisateur à trouver le *fork* qui correspond le mieux à ses attentes. Dans cette évaluation, nous avons sélectionné les 20 premiers projets les plus analysés par `fork-insight`, incluant *linux*, *tensorflow*, *bootstrap*, *spark* ou encore *docker*. Ces projets ont entre 81 (pour *shadowsocks*) et 1855 (pour le jeu *2048*) variantes/*forks* et entre 273 (toujours pour *shadowsocks*) et 3625 (pour *reveal.js*) caractéristiques/mots clés. Les informations concernant le nom

5. https ://github.com/
6. `http://forks-insight.com/`

TABLE 4.9

id	# var.	# car.	100 ét.	temps moy.	treillis	AOC-poset
animate.css	150	323	447	10 ms	366	237 (>1s)
tensorflow	710	2176	380	62 ms	1496	1275 (3s)
bootstrap	1336	2249	515	237 ms	3141	1888 (9s)
angular.js	520	1574	378	129 ms	1073	922 (2s)
linux	581	2303	306	64 ms	1083	1000 (3s)
bitcoin	1014	2829	344	160 ms	1891	1518 (7s)
opencv	385	1489	304	41 ms	716	661 (1s)
spring-boot	234	849	290	20 ms	427	403 (>1s)
react	469	1377	357	328 ms	877	750 (1s)
2048	1855	2428	465	846 ms	4361	2372 (11s)
rails	608	1864	397	75 ms	1272	1130 (3s)
shadowsocks	81	273	295	6 ms	161	145 (>1s)
spark	621	2243	307	62 ms	1240	1095 (3s)
scikit-learn	343	1189	384	34 ms	696	613 (1s)
cgm-rm	156	545	294	13 ms	298	257 (>1s)
docker	196	765	294	22 ms	357	326 (>1s)
oh-my-zsh	669	1301	438	75 ms	1654	1035 (2s)
caffe	748	1786	401	81 ms	1892	1296 (4s)
django	566	1790	376	58 ms	1199	1039 (2s)
reveal.js	1566	3625	515	1723 ms	3552	2523 (15s)

(`id`) et les nombres de variantes (`# var.`) et de caractéristiques (`# car.`) de chacun de ces projets sont données par les trois premières colonnes de la Table 4.9.

Nous avons de plus implémenté les algorithmes présentés précédemment pouvant calculer le voisinage conceptuel dans l'AOC-poset. Le code est disponible en libre accès sur Github [7]. L'implémentation proposée est non optimisée.

QR1 : Afin de vérifier l'applicabilité de cette méthodologie sur les données de `fork-insight`, nous avons d'abord extrait les données des 20 projets sous format `.CSV` grâce à une fonctionnalité proposée par l'outil. Nous avons ensuite conservé uniquement les mots clés correspondant à chaque *fork* et automatisé la transformation de ces listes de mots clés en un unique fichier texte contenant toutes les configurations, à la manière d'un contexte formel. Nous avons ensuite généré pour chaque projet 100 voisinages conceptuels au hasard dans le contexte formel obtenu. Nous avons enregistré le temps de calcul que nous avons ensuite divisé par 100 afin d'obtenir une moyenne en millisecondes du temps de génération d'un voisinage conceptuel : il est présenté dans la colonne `temps moy.` de la Table 4.9.

QR2 : Pour obtenir un ordre de grandeur sur le gain de la génération locale d'AOC-poset par rapport à la génération complète des autres structures conceptuelles, nous avons calculé le nombre de concepts générés pour 100 voisinages conceptuels par rapport aux nombres de concepts des treillis de concepts et AOC-posets complets associés aux mêmes ensembles de données. Ces informations sont consignées respectivement dans les colonnes `100 et.`, `treillis` et `AOC-poset` de la Table 4.9.

Analyse des résultats

QR1 : La colonne *temps moy.* nous révèle que la méthode proposée peut être utilisée dans une application destinée à l'utilisateur. Les temps en moyenne pour calculer un

7. https ://github.com/jcarbonnel/LocGenAOC-poset

voisinage conceptuel sont en dessous de 1 seconde pour la plupart des cas. Exception faite pour `reveal.js`, qui est le plus gros projet en termes de caractéristiques et qui prend en moyenne 1.7 secondes pour déterminer un voisinage conceptuel. Une étude plus approfondie des temps de calcul de ces voisinages montre qu'ils ne sont pas tous similaires. Alors que le voisinage des concepts introduisant peu de configurations se calcule de manière quasi instantanée, le temps de calcul est beaucoup plus long pour les concepts ayant un nombre important de configurations. Cela est dû à deux phénomènes. D'abord, les combinaisons de caractéristiques sont modestes dans `fork-insight` : les intensions des concepts construits à partir de ces données sont donc assez restreintes. Ensuite, certaines des configurations ne présentent qu'un ou deux mots-clés, qui sont partagés par beaucoup d'autres configurations : les concepts introduisant ces mots-clés ont donc une extension très importante. C'est le voisinage de ces concepts-là qui prend le plus de temps de calcul, car beaucoup d'introducteurs d'objets doivent être vérifiés. Les expérimentations montrent que ces cas-là sont cependant assez rares. Une implémentation plus optimisée pourrait réduire ce temps de calcul, même pour ces cas particuliers.

QR2 : La Figure 4.27 montre le nombre de concepts générés pour 100 étapes de génération locale dans l'AOC-poset, l'AOC-poset complet et le treillis des concepts complet pour chaque projet. Les projets sont ordonnés par ordre décroissant sur le nombre de leurs variantes. On peut voir que le nombre de concepts générés dans 100 voisinages conceptuels oscille entre 250 et 500 pour tous les projets. Ce n'est pas le cas pour les nombres de concepts du treillis complets, qui augmentent de manière exponentielle avec le nombre de variantes. L'AOC-poset ne grandit pas aussi vite que le treillis pour les plus gros projets, mais son nombre de concepts reste tout de même très important. Plus le projet comporte de variantes, plus le gain en termes de concepts générés de la génération locale est important. On note cependant que pour les plus petits projets, le nombre de concepts générés à la demande dépasse le nombre de concepts dans le treillis. Les structures conceptuelles étant plus petites, il est plus probable durant les 100 étapes de navigation de repasser plusieurs fois dans la même zone de la structure et donc de générer plusieurs fois les mêmes concepts : dans les petits ensembles de données, il serait donc plus efficace de générer l'AOC-poset complet dès le départ. Nous envisageons d'étudier cette "frontière"dans de futurs travaux. Garder en mémoire un ensemble de concepts déjà générés et proches du concept courant pourrait être une solution pour éviter de générer plusieurs fois les mêmes concepts en revenant sur nos pas durant la navigation.

En résumé (4.4) :

- La **configuration de produits** est une tâche qui consiste à guider un utilisateur dans la sélection d'une variante.
- Son implémentation se réalise généralement à travers un **configurateur** dérivé depuis un **FM cohérent**.
- La **recherche exploratoire** est une stratégie de recherche d'information qui permet à un utilisateur de naviguer dans un espace de recherche en sélectionnant et dé-sélectionnant des mots clés.
- Elle apporte une nouvelle **stratégie complémentaire** aux stratégies actuelles de configuration de produits.
- Les structures conceptuelles sont un support à la recherche exploratoire car elles permettent de structurer l'espace de recherche et d'y naviguer par **navigation conceptuelle**.
- Ainsi, la configuration de produits peut être **appliquée sur des FPLs n'ayant pas de modèles de variabilité**.

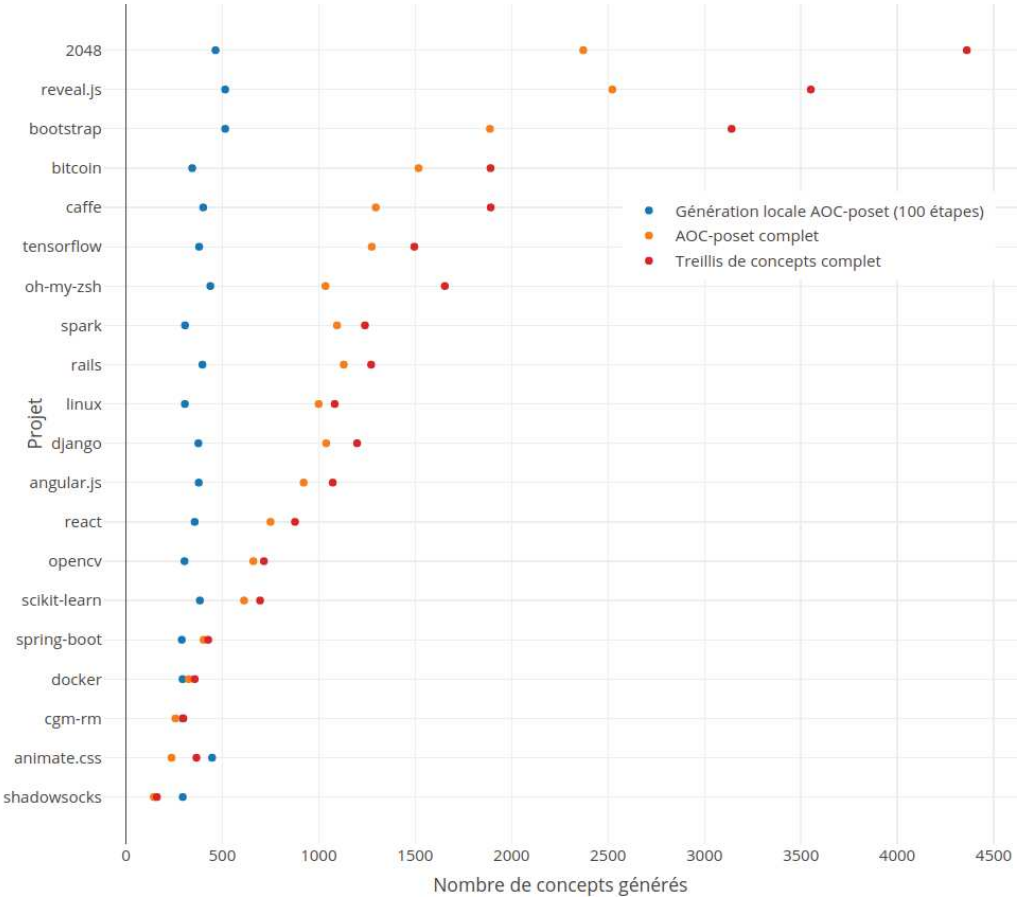


FIGURE 4.27 – Nombre de concepts générés pour chaque projet extrait avec `fork-insight`

— La **génération locale du voisinage conceptuel dans l'AOC-poset** assure le passage à l'échelle de cette stratégie.

4.5 Conclusion

Dans ce chapitre, nous avons illustré le caractère réutilisable de l'AFC à travers la mise en œuvre de trois opérations différentes pour la gestion, la manipulation et l'analyse de la variabilité d'une FPL, dont le support principal n'est autre que les structures conceptuelles. Ceci est illustré en Figure 4.28.

Nous avons d'abord montré comment les informations de variabilité naturellement présentes dans les structures de l'AFC permettent de guider la synthèse de FMs. La place des introducteurs de caractéristiques dans le treillis et les informations livrées par leurs extensions expose les interactions de ces caractéristiques ; cette qualité de l'AFC peut servir à d'autres types de traitements, notamment dans le domaine de l'extraction et de la représentation de l'information : extraction de règles d'implications [RPK11] et de règles d'associations [LS05], de modèles de classes UML [GM93] ou encore d'architectures logicielles [SSS17].

Nous avons ensuite présenté une méthode de composition (union et intersection) basée sur la sémantique de configurations rendue possible par le format d'entrée de l'AFC,

le contexte formel. Le contexte formel permet de travailler sur une représentation en extension de la FPL en manipulant directement les informations des variantes existantes. Modifier, tronquer, compléter, découper ce contexte formel permet de construire des structures spécifiques à certains contextes et donc d'analyser des vues différentes de la variabilité de la famille considérée dans ces cas-là. On pourrait par exemple étudier la variabilité des variantes ayant une caractéristique ou un sous-ensemble de caractéristiques donné, ou bien étudier les nouvelles relations introduites par l'ajout ou la suppression de caractéristiques ou de variantes. Nous avons par exemple défini une intersection approximative, en ne gardant des deux contextes formels de départ que les configurations partielles communes aux deux ensembles. Ces manipulations des contextes formels peuvent servir à produire, ici encore, toutes sortes de modèles [MHCN17]. Notons que le travail inverse (analyser la structure pour modifier le contexte formel) est aussi possible. L'*exploration d'attributs* [GO16] est une technique permettant de guider l'utilisateur dans la construction d'une représentation cohérente de son ensemble de données de départ. Les relations vraies dans le contexte formel sont présentées une à une au concepteur qui peut alors soit valider la relation présentée et passer à l'analyse d'une autre relation, soit l'invalider en ajoutant un contre-exemple (sous la forme d'une nouvelle configuration) dans la table. Cela permet d'avoir un ensemble de configurations représentatif à partir duquel on peut extraire une représentation de la variabilité cohérente et réaliste. En somme, l'AFC permet la manipulation intensionnelle et extensionnelle d'une FPL, et d'assurer la cohérence des deux représentations en propageant les modifications de l'une à l'autre.

Enfin, nous avons appliqué la recherche exploratoire par navigation conceptuelle pour implémenter la configuration de produits. Loin des stratégies basées sur la dérivation de configurateurs, la navigation conceptuelle permet de se déplacer dans l'espace de recherche par étapes minimales depuis une configuration partielle ou valide, c'est-à-dire en découvrant les configurations partielles ou valides les plus proches. C'est une stratégie complémentaire qui, contrairement aux stratégies existantes, ne se concentre pas sur la construction d'une configuration valide, mais sur l'exploration des alternatives les plus proches. Les concepts formels, en représentant toutes les configurations partielles de la FPL et l'ordre de spécialisation qui leur sont associés, permettent de déambuler à travers les abstractions communes des variantes existantes. Ces abstractions communes peuvent aussi servir à des traitements de types réorganisation de modèles de classes UML [Huc15], ou mises en évidence de caractéristiques abstraites pour améliorer les FMs.

Les expérimentations réalisées pour chaque opération pourraient être étendues dans le futur afin de compléter nos observations. Par exemple, nos observations sur la synthèse des FMs sont basées ici sur des matrices de comparaison de produits issues de wikipedia. Évaluer l'aspect des ECFDs obtenus à partir d'autres types de descriptions de produits serait un premier pas dans l'étude de la généralisation de notre approche. Implémenter une interface utilisateur pour encadrer la synthèse pas-à-pas et proposer les différents choix du mapping permettrait de réaliser des études sur la prise en main d'une telle méthode du point de vue d'un utilisateur. Il serait intéressant d'étudier l'intersection approximative plus en détails. Plus spécifiquement, à partir des AC-posets d'union et d'intersection, nous souhaiterions définir des métriques de similarité, basées par exemple sur la taille des intentions ou le nombre de concepts de chaque catégorie (configurations partielles communes et spécifiques). Nous voudrions aussi travailler sur une méthode de composition basée sur les systèmes implicatifs, pour contourner la limite potentielle imposée par la nécessité de calculer toutes les configurations valides. Une comparaison de l'efficacité des deux types d'implémentations est envisageable. Enfin, concernant la recherche exploratoire, nous souhaiterions comparer notre approche avec d'autres approches de sélection de produits. Une étude basée sur des cas d'utilisation et d'un point de vue utilisateur serait aussi très pro-

fitable pour compléter nos présentes observations.

En conclusion, les qualités inhérentes de représentation de la variabilité des structures conceptuelles de l'AFC offrent un support à un panel très large de traitements permettant de faciliter la gestion d'une FPL. Ces qualités s'expriment à travers les nombreuses propriétés de ce cadre (e.g., concepts, place des concepts, introducteurs d'éléments). La construction structurelle de ce support permet la combinaison des traitements pour en établir d'autres plus complexes (e.g., la composition de contextes formels alliés à la synthèse de modèles permet de réaliser de la composition de modèles) qui peuvent jouer un rôle important lors de la migration vers des approches de type LPL.

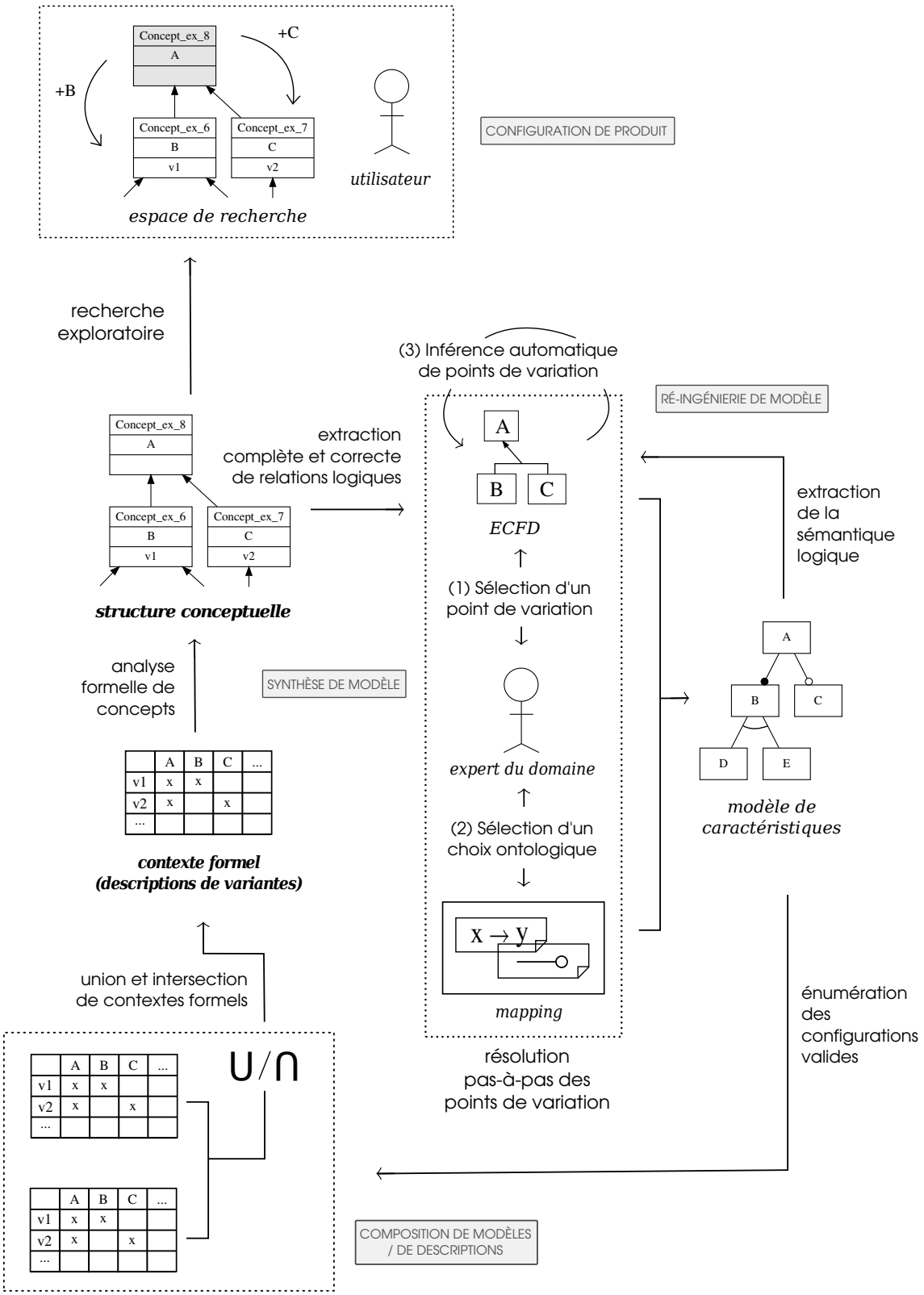


FIGURE 4.28 – L'analyse formelle de concepts : un cadre réutilisable pour la gestion de la variabilité.

Chapitre 5

Les structures de patterns : une extension de l'analyse formelle de concepts pour l'étude de la variabilité étendue

Préambule

Dans le Chapitre 3, nous avons montré que les structures de base de l'analyse formelle de concepts sont des représentations naturelles, structurelles et canoniques de la variabilité. Dans le Chapitre 4, nous avons ensuite montré que l'analyse formelle de concepts est un cadre réutilisable pour la gestion de la variabilité d'une famille de produits logiciels n'ayant pas de modèles de variabilité standards. Dans ce chapitre, nous étudions les structures de patterns, une extension de l'analyse formelle de concepts pour prendre en compte des données plus complexes. Nous montrons comment l'extensibilité de l'analyse formelle de concepts permet d'élargir les problématiques traitées jusqu'ici à de la variabilité plus complexe. Nous étudions pour cela deux extensions des modèles de caractéristiques, qui sont les modèles avec attributs et les modèles avec cardinalités, et nous caractérisons la variabilité étendue qu'ils représentent. Nous définissons ensuite les structures de patterns et comment structurer des données multivaluées pour les traiter avec cette extension. Nous reprenons notre méthode d'extraction d'informations de variabilité booléenne définie au Chapitre 3 et nous montrons comment la réutiliser dans le cadre de la variabilité étendue.

Sommaire

5.1 Introduction

Dans les chapitres précédents, nous avons présenté des modèles de variabilité classiques, ou booléens. Leur but est de représenter les interactions entre des caractéristiques booléennes qui qualifient les produits d'une même famille. Cependant, l'essor important des systèmes à logiciel prépondérant et des systèmes-de-systèmes marque l'apparition de systèmes logiciels de plus en plus complexes. Cela impacte les approches traditionnelles de l'ingénierie des LPLs et engendre le domaine de *l'ingénierie des LPLs complexes* [HGR12], auquel viennent s'ajouter de nouveaux défis incluant la modélisation de la variabilité. Dans ce contexte, les limites de l'expressivité des FMs booléens ont été mises en avant et des extensions furent proposées pour compléter les modèles initiaux et leur permettre de caractériser des LPLs plus complexes. Parmi les extensions enrichissant les FMs, trois d'entre elles cherchent à décrire plus finement la variabilité : la première permet d'ajouter des cardinalités sur les caractéristiques [CBUE02], la deuxième des cardinalités sur les groupes de caractéristiques [RBSP02, Cza99] et la troisième permet d'associer des attributs multivalués aux caractéristiques [CBUE02]. On appelle alors ces modèles des *FMs étendus*. D'autres modèles de variabilité, tels que OVM [PBvdL05] ou CVL [HWC13] proposent des méthodes similaires pour modéliser des informations de *variabilité étendue*, i.e., qui ne sont pas représentées par les FMs booléens.

Les méthodes de ré-ingénierie sont aussi impactées par la complexification des systèmes : cela inclut les méthodes d'extraction de la variabilité. Dans la littérature, les méthodes d'extraction de la variabilité se focalisent sur les FMs booléens [ACP+12, RPK11, CW07, AMHS+14, HLE11, HLE13, DDH+13, LLG+15, LLE14]. Seuls les travaux de Bécan et al. [BBGA15] ont cherché récemment à synthétiser des FMs plus complexes, ici contenant des attributs multivalués.

Dans ce chapitre, nous nous intéressons à l'extraction d'informations de *variabilité étendue*, c'est-à-dire prenant en compte d'autres éléments que des caractéristiques booléennes. Nous nous concentrons sur la variabilité induite par les attributs multivalués ainsi que les deux types de cardinalités. De ce fait, nous devons traiter des descriptions contenant, en plus des caractéristiques booléennes, des valeurs numériques ainsi que des ensembles de littéraux. Nous cherchons à extraire les informations concernant la variabilité sous forme de relations logiques, de manière indépendante des types de modèles de variabilité et de leur éventuelle sémantique ontologique. Cette tâche est une extension des travaux d'extraction de variabilité booléenne présentés en Section 3.4. L'extraction de variabilité étendue est un premier pas dans le processus de rétro-ingénierie de modèles de variabilité dans le domaine des LPLs complexes.

Les attributs multivalués et les cardinalités peuvent introduire un nombre potentiellement élevé de valeurs possibles, qui vont être de plus très spécifiques à chaque produit logiciel. Dans ces cas, les informations concernant la variabilité étendue peuvent devenir très spécialisées et donc très abondantes. Nous cherchons donc aussi à minimiser le nombre de relations extraites afin de faciliter leur analyse par un expert, ou bien leur traitement automatique par un autre processus. Réaliser des groupes de valeurs pertinents et prendre en compte ces groupes dans le processus d'extraction de la variabilité étendue semble être une bonne solution pour réduire le nombre des relations extraites [GM93]. En effet, cela permet de factoriser des relations concernant un ensemble de valeurs spécifiques, en un nombre réduit de relations concernant cette fois des groupes de valeurs plus généraux. De plus, cela permet d'extraire de nouvelles relations qu'il n'était pas possible d'extraire sans considérer ces groupes. Les valeurs numériques et les ensembles de littéraux sont des descripteurs qui peuvent être structurés par spécialisation dans une hiérarchie, à la manière d'une taxonomie. Nous cherchons donc un moyen de traiter des descriptions ayant des

caractéristiques dont les domaines de valeurs sont plus étendus que $\{true, false\}$ et sur lesquels une taxonomie peut être établie. L'expression *taxonomie de valeurs*, initialement utilisée dans ce contexte par Godin et Mili [GM93], se réfère à une structure organisant un ensemble de valeurs par spécialisation, que nous détaillerons plus formellement dans la suite de ce chapitre.

Afin de représenter et de manipuler ces types de descriptions plus complexes, puis d'en extraire des informations sur la variabilité étendue, nous avons étudié les *structures de patterns* [GK01], représentant une formalisation des travaux de Godin et Mili [GM93] bien qu'elle ne se repose pas sur ces travaux. Les structures de patterns forment un cadre mathématique plus général que l'AFC vue aux chapitres précédents, car elles permettent de manipuler des descriptions d'objets complexes qui sont structurées dans des taxonomies. De manière analogue à l'AFC, les structures de patterns produisent des structures représentant un groupement hiérarchique des données d'entrée. De ces structures peuvent être extraites des relations logiques sur les descripteurs des données, pouvant ainsi caractériser la variabilité d'une famille d'objets avec d'autres types de données que des caractéristiques booléennes. Parce que les structures de patterns sont une extension de l'AFC, la plupart des opérations pouvant être appliquées sur des structures conceptuelles de l'AFC traditionnelle peuvent être appliquées de la même façon sur les structures étendues.

Nous avons vu précédemment pourquoi l'AFC est un cadre structurel de représentation de la variabilité des FMs booléens et comment ce cadre pouvait être réutilisé pour mettre en place des opérations de gestion de la variabilité. Ici, nous discutons de son extensibilité, en montrant comment les structures de patterns, une extension du cadre initial pour prendre en compte des données plus complexes, permettent d'étudier et de gérer de nouvelles problématiques. Dans ce chapitre, nous :

- caractérisons les relations logiques correspondant aux FMs étendus (Section 5.2) ;
- définissons formellement les structures de patterns, en illustrant sur un exemple simple chacune des définitions (Section 5.3) ;
- proposons une méthode pour transformer une FPL décrite par des caractéristiques booléennes et multivaluées en une représentation permettant de les traiter par ce cadre mathématique (Section 5.4) ;
- montrons comment extraire les informations caractérisant la variabilité étendue considérées ici et proposons des lignes directives pour en éliminer la redondance en considérant les taxonomies de valeurs (Section 5.5) ;
- mettons en place une série d'expérimentations pour évaluer l'applicabilité, l'employabilité et la pertinence de l'approche proposée, ainsi que le gain de l'utilisation des taxonomies de valeurs sur le nombre de relations extraites (Section 5.6).

Les sections de ce chapitre suivent les différentes étapes du processus d'extraction des connaissances, tel qu'illustré dans la Figure 5.1.

5.2 Caractérisation de la variabilité étendue

Dans cette section, nous allons considérer l'exemple du FM étendu présenté à la Figure 5.2. Ce modèle représente des applications de commerce électronique : il est similaire à celui présenté précédemment, mais il comporte des informations additionnelles qui ne sont pas représentables dans les FMs booléens traditionnels (apparaissant en gras dans la figure) :

- la caractéristique `catalog` possède un attribut `productsPerPage` de type entier spécifiant le nombre maximum de produits pouvant être affichés dans une page du catalogue ;

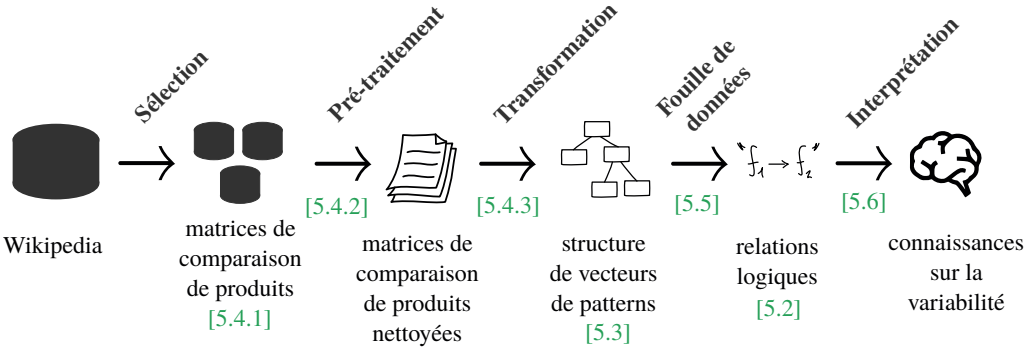


FIGURE 5.1 – Différentes étapes du processus d'extraction des connaissances en fonction des sections du chapitre

— la caractéristique `catalog` possède aussi une cardinalité $[1..n]$ statuant qu'une application de commerce électronique a forcément un catalogue, mais qu'elle peut en avoir plusieurs à la fois ;
— les deux groupes de caractéristiques sous `catalog` et `payment_method` sont contraints par une cardinalité déterminant le nombre minimum et maximum de caractéristiques qui peuvent être sélectionnées dans le groupe ;
— la contrainte *cross-tree* $\texttt{grid} \rightarrow \texttt{productsPerPage} \geq \texttt{15}$ est une implication entre une caractéristique et une valeur d'attribut.

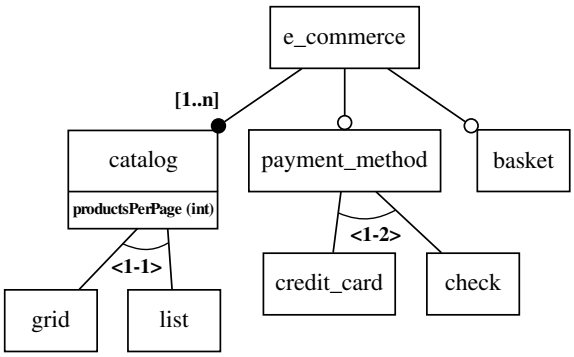


FIGURE 5.2 – Exemple de FM étendu avec une cardinalité de caractéristique, deux cardinalités de groupes et un attribut

Ces informations correspondent à deux extensions des FMs booléens ; elles permettent d'en améliorer l'expressivité. Nous allons présenter ces deux extensions et identifier la sémantique logique correspondant aux nouveaux types de contraintes de variabilité qu'elles introduisent, i.e., non présentes dans les modèles booléens.

5.2.1 Attributs multivalués

La première extension améliorant l'expressivité d'un FM permet d'ajouter des attributs multivalués aux caractéristiques. Elle permet de modéliser des informations plus détaillées sans complexifier le modèle [CBUE02]. En effet, si l'on représente un modèle avec uniquement des caractéristiques, chaque valeur d'un attribut serait représentée par une caractéristique booléenne. Dans les cas où les valeurs possibles d'un attribut sont nombreuses (pour des valeurs numériques par exemple), le nombre de caractéristiques deviendrait alors

trop important et le modèle rapidement incompréhensible. Par exemple, introduire l'attribut `productsPerPage` de type entier dans la caractéristique `catalog` réduit le nombre de caractéristiques nécessaires pour représenter la même information, comme montré dans la Figure 5.3.

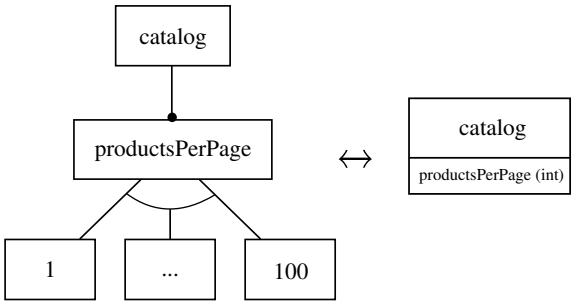


FIGURE 5.3 – Représenter des valeurs avec des caractéristiques booléennes (gauche) ou des attributs (droite)

Nous ferons la distinction entre "attribut multivalué", indiquant un attribut ayant plusieurs valeurs possibles et "valeurs uniques / multiples pour un attribut", indiquant qu'un attribut peut avoir une seule valeur, ou plusieurs valeurs pour un même produit, respectivement. Un attribut multivalué peut donc être soit à valeurs uniques, soit à valeurs multiples ; dans notre exemple, `productsPerPages` est un attribut multivalué à valeurs uniques, car il peut avoir une seule valeur dans $\{1..100\}$ pour chaque produit.

Ajouter des attributs multivalués dans les FMs permet d'exprimer des contraintes de variabilité plus complexes, telles que des implications et des exclusions mutuelles comprenant des valeurs d'attributs. Comme le laissent sous-entendre leurs noms, les groupes de caractéristiques et l'arbre des caractéristiques ne sont définis que sur l'ensemble des caractéristiques et ne sont donc pas impactés par les attributs. Les contraintes de variabilité rajoutées par cette extension correspondent donc aux relations logiques suivantes : implications, cooccurrences et exclusions mutuelles comprenant au moins une valeur d'attribut. Nous appelons ces relations logiques additionnelles des contraintes de *variabilité étendue*.

5.2.2 Cardinalités

La seconde extension améliorant l'expressivité des modèles introduit des cardinalités sur les caractéristiques et les groupes [CHE04].

Les cardinalités sur les groupes de caractéristiques définissent le nombre minimum et maximum de sous-caractéristiques qui peuvent être sélectionnées dans un groupe et sont notées $\langle min - max \rangle$. Ainsi, la notation graphique des groupes *xor* et *or* des modèles booléens n'a plus lieu d'être : les groupes *xor* sont définis par la cardinalité $\langle 1 - 1 \rangle$ et les groupes *or* par $\langle 1 - n \rangle$. Cela peut être exprimé en logique des propositions par la conjonction de chaque combinaison de sous-caractéristiques du groupe qui est autorisée par la cardinalité. Soit p une caractéristique et $\{f_1, f_2, f_3\}$ un groupe de caractéristiques avec p comme parent et associé à la cardinalité $\langle 2 - 3 \rangle$. La formule logique représentant ce groupe est :

$$(f_1 \rightarrow p) \wedge (f_2 \rightarrow p) \wedge (f_3 \rightarrow p) \wedge$$
$$p \rightarrow ((f_1 \wedge f_2 \wedge \neg f_3) \vee (f_1 \wedge f_3 \wedge \neg f_2) \vee (f_2 \wedge f_3 \wedge \neg f_1) \vee (f_1 \wedge f_2 \wedge f_3))$$

La cardinalité sur les groupes permet de considérer que les deux types de groupes représentent en réalité le même type de contraintes de variabilité ; non plus en tant que groupe *xor* ou *or*, mais en tant que groupe associé à une cardinalité. Les cardinalités sur les

groupes n'ajoutent pas de nouveaux types de contraintes de variabilité, mais en généralisent deux existantes.

Les cardinalités sur les caractéristiques définissent le nombre minimum et maximum d'occurrences (aussi appelées clones) d'une caractéristique dans une configuration valide du modèle et sont notées $[min..max]$. Ces cardinalités peuvent être vues comme des attributs : dans l'exemple de la Figure 5.2, la cardinalité de `catalog` peut être représentée par un attribut `occurrence` avec le domaine $\{1, 2, \ldots, n\}$ sans que l'on perde d'information. Par conséquent, les cardinalités sur les caractéristiques peuvent aussi être représentées par des caractéristiques, comme vu précédemment avec les valeurs d'attributs. Il est à noter que malgré le fait que la notation soit différente, l'information reste exactement la même. La Figure 5.4 illustre les différentes (mais équivalentes) manières de les représenter.

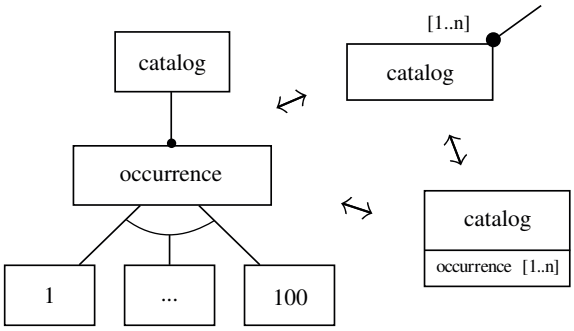


FIGURE 5.4 – Différentes manières de représenter une cardinalité de caractéristique

Les types de relations logiques induites par ces cardinalités sont donc exactement les mêmes que pour les attributs, i.e., les contraintes de *variabilité étendue*.

La grammaire des contraintes de variabilité représentant les FMs et ses deux extensions améliorant leur expressivité est présentée en Figure 5.5. Par souci de lisibilité, nous simplifions la formule logique représentant les groupes et leur cardinalité par la notation suivante : $(p, \{f_1, \ldots, f_n\}, \langle min, max \rangle)$.

contraintes de		
variabilité étendue	:=	*relation**
relation	:=	*implication* | *co-occurrence* |*mutex* | *groupe*
implication	:=	*élément* '→' *élément*
co-occurrence	:=	*élément* '↔' *élément*
mutex	:=	*élément* '→' '¬' *élément*
groupe	:=	'(' *caractéristique* ','
		'{' *ensemble_caractéristiques* '}' ',' *cardinalité* ')'
élément	:=	*caractéristique* | *attribut*
ensemble_caractéristiques	:=	*caractéristique* |
		ensemble_caractéristiques ',' *caractéristique*
attribut	:=	**nom_attribut** '=' **valeur**
caractéristique	:=	**nom_caractéristique**
cardinalité	:=	'⟨' **min** ',' **max** '⟩'

FIGURE 5.5 – Grammaire des contraintes de variabilité étendue

En résumé (5.2) :

— On étudie deux extensions des FMs permettant d'**améliorer leur expressivité**
— Elles permettent de prendre en compte des **attributs multivalués** et des **cardinalités**
— Les **cardinalités sur les caractéristiques** peuvent être représentées par des **attributs multivalués numériques**
— Les **cardinalités sur les groupes** généralisent les notations des groupes *or* et *xor*
— Ces extensions rajoutent à la sémantique logique des modèles booléens traditionnels, **des implications, cooccurrences et mutex incluant (au moins) une valeur d'attribut**

Par la suite, nous étudions les structures de patterns, une généralisation de l'AFC prenant en compte des descriptions plus complexes que des ensembles de caractéristiques booléennes, i.e., des caractéristiques multivaluées dont le domaine de valeurs est organisé sous forme de taxonomies. Les structures de patterns permettent d'extraire des relations logiques correspondant aux contraintes de variabilité étendue que nous venons de caractériser. De plus, elles permettent de mettre en évidence des relations entre des valeurs plus générales (représentant des groupes de valeurs de même type) que les valeurs initiales.

5.3 Les structures de patterns : définitions

Dans cette section, nous définissons les structures de patterns, d'abord de manière générale, puis de façon plus théorique. Nous verrons qu'elles représentent des taxonomies de valeurs d'un même type, constituant des descriptions d'objets complexes. Puis, nous introduisons les vecteurs de structures de patterns comme étant un moyen de composer plusieurs structures de patterns de types différents, afin de représenter des descriptions d'objets constituées de plusieurs valeurs de types hétérogènes.

5.3.1 Généralités

Les *structures de patterns* [GK01] permettent de généraliser l'AFC afin de prendre en compte des descriptions d'objets plus complexes que des ensembles d'attributs booléens. Elles sont une forme spécialisée des correspondances de Galois [BM70]. On trouve par exemple dans la littérature des descriptions prenant la forme de graphes représentant des molécules [GK01], de triplets RDF [RATN17, BCNR17], ou encore de séquences [BEJ+16]. Chaque objet d'un ensemble O est décrit par un pattern (ou description) d'un ensemble noté $\mathcal{D}$. Les patterns de $\mathcal{D}$ peuvent être de n'importe quel type de données, sous réserve de pouvoir définir un *opérateur de similarité* (noté $\sqcap$) sur ces patterns. Un opérateur de similarité, appliqué à un ensemble de patterns de $\mathcal{D}$ renvoie un pattern de $\mathcal{D}$ représentant la similarité, ou généralisation, de ses arguments. $(\mathcal{D}, \sqcap)$ doit former un demi-treillis supérieur (ou sup-demi-treillis), i.e., une structure dans laquelle chaque sous-ensemble d'éléments possède une borne supérieure étant un élément de la structure ; les éléments y sont donc organisés par spécialisation/généralisation. Par la suite, et afin de s'aligner sur les travaux de Godin et Mili [GM93], nous utiliserons le terme *taxonomie* pour faire référence à un demi-treillis supérieur organisant un ensemble de valeurs d'un même type où chaque ensemble de valeurs possède donc une borne supérieur étant une valeur de même type représentant la similarité de l'ensemble de départ.

Pour illustrer les définitions suivantes, considérons l'exemple de la Table 5.1, représentant 4 objets (lignes), chacun décrit par un intervalle de valeurs.

TABLE 5.1 – 4 objets décrits par des intervalles de valeurs

O	intervalles
o_1	[2,4]
o_2	[6,7]
o_3	[3,5]
o_4	[2,5]

Dans cet exemple, chaque intervalle de valeurs est considéré comme un pattern d'un ensemble $\mathcal{D}_{inter}$. Seulement, pour que l'on puisse les traiter avec ce cadre mathématique, on doit pouvoir définir la similarité entre deux de ces intervalles. Une solution proposée par Kaytoue et. al. est de considérer le plus petit intervalle contenant les deux intervalles en arguments [KAMN10]. On définit alors l'opérateur de similarité $\sqcap_{inter}$ par :

$$[a_1, b_1] \sqcap_{inter} [a_2, b_2] = [min(a_1, a_2), max(b_1, b_2)]$$

Dans notre exemple, on a donc [3,5] $\sqcap_{inter}$ [6,7] = [3,7] et l'intervalle [3,7] ainsi obtenu est aussi un pattern de $\mathcal{D}_{inter}$. L'ensemble des patterns issus de la table est donc inclus dans $\mathcal{D}_{inter}$, mais $\mathcal{D}_{inter}$ possède aussi d'autres patterns non présents dans la table.

Un opérateur de similarité est associé à une relation de subsomption $\sqsubseteq$ permettant d'organiser les patterns par spécialisation dans une taxonomie :

$$a \sqsubseteq b \iff a \sqcap b = a, \qquad \forall a, b \in \mathcal{D}$$

Toujours dans notre exemple, on a alors [3,7] $\sqsubseteq_{inter}$ [6,7] et [3,7] $\sqsubseteq_{inter}$ [3,5]. La Figure 5.6 présente l'organisation des patterns issus de la Table 5.1 en fonction de la relation de subsomption associée à l'opérateur de similarité $\sqcap_{inter}$, que l'on notera $(\mathcal{D}_{inter}, \sqcap_{inter})$. Chaque encadré représente un pattern de $\mathcal{D}_{inter}$ et les flèches indiquent la relation de subsomption. En gras sont spécifiés les 4 patterns provenant de la description de la Table 5.1 ; les autres patterns sont ceux obtenus par application de l'opérateur de similarité $\sqcap_{inter}$.

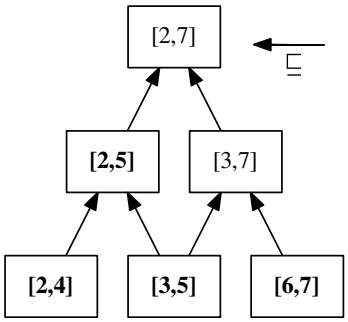


FIGURE 5.6 – Organisation des patterns issus de la Table 5.1 en fonction de la relation de subsomption associée à l'opérateur de similarité $\sqcap_{inter}$. En gras sont représentés les patterns présents dans la description initiale, les autres sont obtenus par application de l'opérateur de similarité

Notons que dans une structure conceptuelle, les "plus grands intervalles" appraîtront dans les concepts "en haut" de la structure, car ils correspondent aux descripteurs couvrant le plus d'objets.

On peut introduire une valeur de *dissimilarité* dans la taxonomie, généralement notée *, pour indiquer que deux éléments ne sont pas similaires.

5.3.2 Définitions

Cette généralisation de l'AFC prend en entrée, non plus un contexte formel, mais une *structure de patterns*.

Définition 5.1 (Structure de patterns [GK01])**.** *Une structure de patterns est un triplet* $PS = (O, (\mathcal{D}, \sqcap), \delta)$ *où* O *est un ensemble d'objets,* $(\mathcal{D}, \sqcap)$ *est un sup-demi-treillis représentant une taxonomie de patterns et* $\delta : O \mapsto \mathcal{D}$ *est un mapping associant chaque objet de* O *à un pattern de* $\mathcal{D}$*.*

Si les patterns de $\mathcal{D}$ sont des ensembles d'attributs binaires et que l'opérateur de similarité $\sqcap$ est défini par l'intersection ensembliste $\cap$, alors on retombe dans le cadre de l'AFC traditionnelle.

L'opérateur de similarité et le sup-demi-treillis de patterns qui en découle garantissent la conservation des propriétés des opérateurs α et β de l'analyse formelle de concepts. Afin de les généraliser à tout type de patterns, on redéfinit $\alpha : 2^O \mapsto \mathcal{D}$ et $\beta : \mathcal{D} \mapsto 2^O$ de la manière suivante :

$$\begin{aligned} \alpha(R) &= \prod_{o \in R} \delta(o), && \text{avec } R \subseteq O \\ \beta(d) &= \{o \in O \mid d \sqsubseteq \delta(o)\}, && \text{avec } d \in (D, \sqcap) \end{aligned}$$

L'opérateur α associe à tout sous-ensemble d'objets $R \in O$ le pattern le plus spécifique de $(\mathcal{D}, \sqcap)$ décrivant tous les objets de R. L'opérateur β associe à tout pattern $d \in (\mathcal{D}, \sqcap)$ l'ensemble des objets de O correspondant à ce pattern, c'est-à-dire tous les objets étant soit décrits par d, soit décrits par un pattern qui subsume d. Dans notre précédent exemple, $\alpha(\{o_1, o_2\}) = \delta(o_1) \sqcap_{inter} \delta(o_2) = [2, 4] \sqcap_{inter} [6, 7] = [2, 7]$ et $\beta([2, 7]) = \{o_1, o_2, o_3, o_4\}$.

L'application des deux opérateurs α et β sur une structure de patterns extrait un ensemble fini de *concepts de patterns*, noté C_{ps}. Un concept de patterns représente un ensemble maximal d'objets de O et le pattern le plus spécifique de la taxonomie $(\mathcal{D}, \sqcap)$ correspondant à tous ces objets.

Définition 5.2 (Concept de patterns)**.** *Un concept d'une structure de patterns* $PS = (O, (\mathcal{D}, \sqcap), \delta)$ *est une paire* $C = (R, d) \in C_{PS}$ *telle que* $R \subseteq O$ *et* $d \in (\mathcal{D}, \sqcap)$*, vérifiant* $\alpha(R) = d$ *et* $\beta(d) = R$*.* R *est appelé l'extension et* d *l'intension du concept.*

Les paires ($\{o_1, o_2, o_3, o_4\}$, [2,7]) et ($\{o_2, o_3\}$, [3,7]) sont deux concepts de notre exemple.

De la même manière que pour l'AFC traditionnelle, on définit un ordre partiel sur l'ensemble des concepts de patterns C_{PS} appartenant à une structure de patterns PS, basé sur l'inclusion ensembliste de leurs extensions.

Définition 5.3 (Ordre de spécialisation $\leq_{ps}$)**.** *Soient deux concepts de patterns* $C_1 = (O_1, d_1)$ *et* $C_2 = (O_2, d_2)$ *d'une structure de patterns.* $C_1 \leq_{ps} C_2$ *si et seulement si* $O_1 \subseteq O_2$ *et* $d_2 \sqsubseteq d_1$*.* C_1 *est appelé sous-concept de* C_2 *et* C_2 *super-concept de* C_1*.*

On dit que les concepts sont organisés par spécialisation, car un concept du treillis est décrit par un pattern plus spécialisé que les patterns décrivant ses super-concepts. Par exemple, ($\{o_2, o_3\}$, [3,7]) est un sous-concept de ($\{o_1, o_2, o_3, o_4\}$, [2,7]) et on a bien $[2, 7] \sqsubseteq [3, 7]$ dans $(\mathcal{D}_{inter}, \sqcap_{inter})$.

L'ensemble de tous les concepts C_{PS} d'une structure de patterns PS muni de $\leq_{ps}$ forme un treillis des concepts de patterns $(C_{ps}, \leq_{ps})$.

Définition 5.4 (Treillis des concepts de patterns)**.** *Soit une structure de patterns* $PS = (O, (\mathcal{D}, \sqcap), \delta)$ *et* C_{PS} *son ensemble de concepts associés. L'ensemble* C_{PS} *muni de l'ordre partiel* $\leq_{ps}$ *forme une structure de treillis appelée treillis des concepts de patterns.*

Le treillis des concepts de patterns (ou plus simplement treillis des patterns) associé aux descriptions de la Table 5.1 est présenté dans la Figure 5.7.

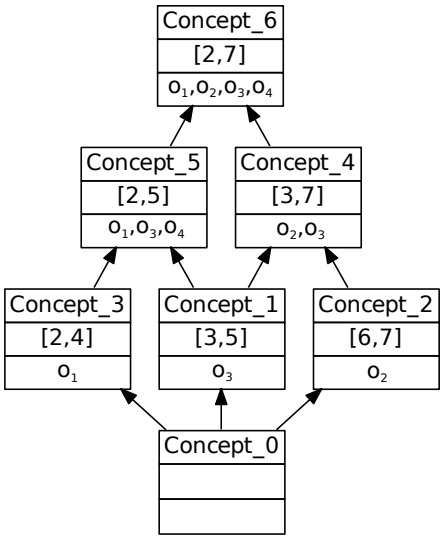


FIGURE 5.7 – Treillis des patterns associé à la Table 5.1

5.3.3 Les vecteurs de patterns

Jusqu'ici, nous avons illustré les structures de patterns en utilisant des patterns ayant un type atomique, tel que les intervalles de valeurs. Cependant, la condition nécessaire pour qu'un type de données puisse être traité comme une structure de patterns (i.e., posséder un opérateur de similarité) laisse beaucoup de liberté quant à la définition d'un "type de patterns". Par exemple, il est possible de combiner plusieurs types de patterns pour obtenir un *vecteur de patterns* de la forme $\langle d_1, d_2, \ldots, d_n \rangle$, où d_i, $i \in \{1, 2, \ldots, n\}$ est un pattern d'une taxonomie composante $(\mathcal{D}_i, \sqcap_i)$. Chaque taxonomie composante peut être d'un type différent de celui des autres taxonomies composant le vecteur. On peut définir un opérateur de similarité $\sqcap_{vp}$ entre vecteurs de patterns composés des mêmes taxonomies composantes dans le même ordre $(\mathcal{D}_1, \sqcap_1), (\mathcal{D}_2, \sqcap_2), \ldots, (\mathcal{D}_n, \sqcap_n)$ par :

$$\langle d_1^i, d_2^i, \ldots, d_n^i \rangle \sqcap_{vp} \langle d_1^j, d_2^j, \ldots, d_n^j \rangle = \langle d_1^i \sqcap_1 d_1^j, d_2^i \sqcap_2 d_2^j, \ldots, d_n^i \sqcap_n d_n^j \rangle$$

Ces vecteurs de patterns sont donc aussi des patterns et peuvent être traités de la même manière que les patterns de type atomique vus précédemment. Pour simplifier la terminologie, nous appellerons *structure de vecteurs de patterns* une structure de patterns composées de vecteurs de patterns

Ces vecteurs de patterns apparaissent comme de bons candidats pour caractériser des objets ayant des descriptions hétérogènes tels que les systèmes logiciels. Un exemple de leur utilisation dans ce contexte sera donné dans la section suivante.

En résumé (5.3) :

- Les **structures de patterns** permettent de généraliser l'AFC pour prendre en compte des descriptions complexes appelées **patterns**
- On doit pouvoir définir la similarité (ou généralisation) d'un ensemble de patterns $\mathcal{D}$ d'un même type via un **opérateur de similarité** $\sqcap$
- Un ensemble de patterns est alors organisé sous la forme d'une **taxonomie** notée $(\mathcal{D}, \sqcap)$, qui a la forme d'un sup-demi-treillis
- On peut combiner plusieurs taxonomies différentes pour former des **vecteurs de patterns**, qui sont aussi des patterns
- Les vecteurs de patterns vont nous être utiles pour représenter des **descriptions complexes et hétérogènes** de systèmes logiciels

5.4 Structuration de descriptions de produits multivalées

Nous nous intéressons ici à des descriptions *multivaluées* de produits logiciels, c'est-à-dire donnant des informations plus riches que la présence ou non d'une *feature* dans un produit. Ces informations peuvent par exemple prendre la forme de valeurs numériques, littérales, ou bien des ensembles de valeurs.

Nous avons précédemment représenté une famille de produits et ses *features* sous la forme d'une matrice booléenne, ou contexte formel dans le cadre de l'AFC. Cette matrice indique par une croix dans une cellule si le produit correspondant à la ligne possède la *feature* correspondant à la colonne. Pour permettre une description plus complexe des produits, il est nécessaire de prendre en compte des valeurs autres que booléennes dans les cellules. C'est donc à partir de matrices multivaluées, ou contextes multivalués, que nous allons travailler.

Dans cette section, nous présentons d'abord les matrices de comparaison de produits, qui sont des matrices multivaluées offrant aux utilisateurs un formalisme de comparaison de produits similaires. Ensuite, nous définissons un format pour ces matrices afin qu'elles puissent être traitées automatiquement. Enfin, nous présentons une méthode pour transformer une matrice multivaluée en structures de patterns.

Cette section correspond aux étapes de prétraitement et de transformation (ou nettoyage) des données dans le processus d'extraction de connaissances.

5.4.1 Les matrices de comparaison de produits, une source abondante de descriptions complexes de produits logiciels

Les matrices de comparaison de produits (MCPs) [SAB13, BSA+14] sont des tables représentant un ensemble de produits en fonction d'un certain nombre de leurs caractéristiques. Elles décrivent un ensemble de produits de la même famille de façon à faciliter leur comparaison par un utilisateur. Les Figures 5.8, 5.9 et 5.10 présentent des exemples de MCPs sur des logiciels de contrôle de versions issus de Wikipedia [1]. Il y a plusieurs raisons qui nous motivent à nous servir de matrices de comparaison de produits comme support pour l'extraction de variabilité expressive. Tout d'abord, elles rassemblent dans leurs cellules des types de données hétérogènes, comprenant des attributs booléens (ayant des valeurs vrai/faux et pouvant être considérés comme des *features*, voir Figure 5.8) et des

1. https ://en.wikipedia.org/wiki/Comparison_of_version_control_software, dernier accès en mars 2018

attributs multivalués de types variés (entiers, réels, dates, littéraux, voir Figure 5.9). Ensuite, on trouve un nombre considérable de MCPs sur internet, par exemple sur des sites marchands, sur des sites d'information, ou sur des initiatives telles que Wikipedia. Ces matrices peuvent entre autres représenter des FPLs, comme on peut le voir dans la catégorie *software comparison* de Wikipedia [2]. Enfin, le formalisme des MCPs a déjà été étudié dans la littérature [SAB13, BSA+14] et des outils *open source* tels que `OpenCompare` [3] vont pouvoir nous aider à extraire et à traiter les données.

Software	Atomic commits	File renames	Merge file renames	Symbolic links	Pre-/post-event hooks	Signed revisions	Merge tracking	End of line conversions
AccuRev SCM	Yes	Yes	Partial[nb 15]	Yes	Yes	Yes	Yes	Yes
GNU Bazaar	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes[21]
BitKeeper	Yes	Yes	Yes	Yes	Yes	Unknown	Yes	Yes
CA Software Change Manager	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ClearCase	Partial[nb 16]	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Code Co-op	Yes	Yes	Yes	No	Partial	No	No	No
Codeville	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown
CVS	No	No	No	No	Partial	No	No	Yes
CVSNT	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
darcs	Yes	Yes	Yes	No[nb 17]	Yes	Yes	N/A[nb 18]	No
Dimensions CM	Yes	Yes	Yes	No	Yes	Unknown	Yes	Yes
Fossil	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes[nb 22]
Git	Yes	Partial[nb 23]	Yes	Yes	Yes	Yes[nb 24]	Yes	Yes
GNU arch	Yes	Yes	Unknown	Yes	Yes	Yes	Unknown	Unknown

FIGURE 5.8 – Extrait de MCP sur des logiciels de contrôle de versions présentant des caractéristiques booléennes (en omettant les valeurs "*partial*" et "*unknown*" et les notes en exposant)

Software	Repository model	Concurrency model	License	Platforms supported
AccuRev SCM	Client-server	Merge or lock	Proprietary	Most Java Platforms (Unix-like, Windows, OS X)
GNU Bazaar	Distributed and Client-server	Merge	GNU GPL	Unix-like, Windows, OS X
BitKeeper	Distributed	Merge	Apache	Unix-like, Windows, OS X
ClearCase	Client-server	Merge or lock[nb 1]	Proprietary	Linux, Windows, AIX, Solaris, HP UX, i5/OS, OS/390, z/OS,
Code Co-op	Distributed	Merge	Proprietary	Windows
Codeville	Distributed	precise codeville merge	BSD	Unix-like, Windows, OS X
CVS	Client-server	Merge	GNU GPL	Unix-like, Windows, OS X
CVSNT	Client-server	Merge or lock	GPL or proprietary	Unix-like, Windows, OS X, i5/OS
darcs	Distributed	Merge	GNU GPL	Unix-like, Windows, OS X
Dimensions CM	Client-server	Merge or lock	Proprietary	Windows, Linux, Solaris, AIX, HP UX, z/OS
Endevor	Client-server	Merge or Lock	Proprietary	z/OS
Fossil	Distributed	Merge	BSD	POSIX, Windows, OS X, Other
Git	Distributed	Merge	GNU GPL	POSIX, Windows, OS X
GNU arch	Distributed	Merge	GNU GPL	Unix-like, Windows, OS X

FIGURE 5.9 – Extrait de MCP sur des logiciels de contrôle de versions présentant des caractéristiques multivaluées

Ces MCPs possèdent un inconvénient : elles ne sont que peu, voire pas formalisées, comme le montre l'extrait de la Figure 5.10. De plus, il n'existe à ce jour aucun consensus sur la façon de formater une MCP. De ce fait, certaines MCPs que l'on trouve sur internet possèdent des cellules avec des valeurs ambiguës, partielles, incohérentes ou encore nulles, ce

2. https ://en.wikipedia.org/wiki/Category :Software_comparisons
3. https ://github.com/OpenCompare

qui peut en gêner la compréhension ou l'analyse, qu'elles soient automatiques ou manuelles [SAB13].

Software	History
AccuRev SCM	First publicly released in 2002
GNU Bazaar	Loosely related to baz. Sponsored by Canonical Ltd..
BitKeeper	Influenced by Sun WorkShop TeamWare
CA Software Change Manager	Original company founded in 1977; CA SCM (then called CCC/Harvest) first released in 1995.
ClearCase	Developed beginning in 1990 by Atria Software, following concepts developed by Apollo Computer in DSEE during the 1980s. The most recent version is 9.0.0, released in March 2016.
Code Co-op	The first distributed VCS, demoed in 1997,[75] released soon after.
CVS	First publicly released July 3, 1986; based on RCS
CVSNT	First publicly released 1998; based on CVS. Started by CVS developers with the goal adding support for a wider range of development methods and processes.
darcs	First announced on April 9, 2003
Dimensions CM	Developed by SQL Software under the name "PCMS Dimensions" during the late 1980s (PCMS standing for Product Configuration Management). Through number of company acquisitions the product was released under names "PVCS Dimensions" (1990s, Intersolv), "Dimensions" (early 2000s, Merant), "ChangeMan Dimensions" (2004, Serena Software) and finally "Dimensions CM" (since 2007, Serena Software).
Fossil	Fossil and SQLite have used Fossil since 21 July 2007.
Git	Started by Linus Torvalds in April 2005, following the BitKeeper controversy.[81]
GNU arch	Started by Tom Lord in 2001, it later became part of the GNU project. Lord resigned as maintainer in August 2005.

FIGURE 5.10 – Extrait de MCP sur des logiciels de contrôle de versions présentant des valeurs non formatées

5.4.2 Un format pour garantir l'analyse automatique des matrices de comparaison de produits

Afin de pouvoir traiter ces données de manière automatique, nous avons défini des règles de formatage. Ces règles sont basées sur les travaux de Sannier et al. [SAB13] et de Bécan et al. [BSA+14], qui ont étudié la formalisation des matrices de comparaison de produits. Nous avons identifié trois aspects d'une MCP qu'il est nécessaire de contrôler lors de son formatage : l'agencement, les valeurs des cellules et la cohérence globale.

Agencement d'une MCP

Il est possible de trouver des matrices dans lesquelles certaines cellules sont partagées par plusieurs lignes ou par plusieurs colonnes. Par la suite, nous travaillerons uniquement avec des matrices dans lesquelles l'intersection d'une ligne et d'une colonne n'est représentée que par une seule cellule. Cela permettra d'éviter l'ambiguïté causée par le partage de valeurs et facilitera le traitement automatique.

En règle générale, l'ensemble des caractéristiques est décrit dans la ligne du haut et l'ensemble des produits dans la colonne de gauche. On peut cependant trouver des MCPs pour lesquelles cet ordre est inversé. Par la suite, nous garderons la représentation prédominante, là encore pour faciliter le traitement automatique.

Valeurs des cellules d'une MCP

Les travaux de Sannier et al. [SAB13] et de Bécan et al. [BSA+14] étudient les MCPs brutes extraites de Wikipedia et mettent en évidence 7 types de cellules :

— les cellules **booléennes**, qui possèdent des valeurs telles que vrai/faux ou oui/non. Ces valeurs indiquent si le produit possède la caractéristique ou non. Il n'y a donc qu'une valeur unique pour un produit ;

— les cellules **monovaluées**, donnant une valeur unique pour une caractéristique et un produit (avec un domaine différent de celui des cellules booléennes) ;
— les cellules **multivaluées**, donnant plusieurs valeurs pour une caractéristique et un produit ;
— les cellules ayant des valeurs **partielles ou conditionnelles**, donnant une (ou des) valeur(s) pour une caractéristique, mais sous certaines conditions ;
— les cellules ayant des valeurs **inconnues**, indiquant clairement que l'information n'est pas connue ;
— les cellules **vides** ;
— les cellules **incohérentes**, donnant des valeurs qui ne sont pas en adéquation avec les autres valeurs possibles de la caractéristique.

Pour extraire de la variabilité expressive, nous considérons des FPLs décrites par des *features* ainsi que par des attributs multivalués. Les *features* peuvent être vues comme des attributs ayant des valeurs booléennes. Les attributs multivalués peuvent avoir des ensembles de valeurs de type entier, réel ou littéral (i.e., non numérique). De plus, comme nous utilisons le formalisme des structures de patterns, nous remplaçons les valeurs partielles, inconnues, vides et incohérentes des MCPs brutes par la valeur "*", qui définit l'absence d'information dans ce cadre. De ce fait, même si l'absence d'information dénote un problème dans les données de départ, elles pourront être traitées par les structures de patterns.

Par conséquent, une cellule d'une **MCP bien formée** pourra contenir :
— soit une **valeur booléenne**, indiquant si le produit possède la caractéristique ou pas. Ces caractéristiques sont donc considérées comme des *features*.
— soit un **ensemble non-vide de valeurs**, représentant la (ou les) valeur(s) de la caractéristique. Ces caractéristiques sont alors considérées comme des attributs multivalués avec un domaine de type entier, réel ou littéral. Les valeurs sont séparées par des points virgules.
— la valeur **"*"** représentant l'absence d'information.

Cohérence globale de la MCP

La dernière problématique liée au traitement automatique des matrices de comparaison réside dans le fait que certaines cellules peuvent contenir les mêmes informations mais présentées de manières différentes. Dans ce cas, il est important d'utiliser les mêmes termes, ou mots-clé pour représenter les mêmes concepts, pour qu'ils puissent être considérés comme équivalents lors du traitement automatique, sans avoir recours à des techniques de traitement du langage naturel. Par exemple, la colonne `Platforms supported` de la matrice de la Figure 5.9 donne la valeur `Unix-like` pour certains produits et `Linux` et `Solaris` pour d'autres.

La Table 5.2 présente un extrait des MCPs des Figures 5.9 et 5.10 respectant le format défini. La caractéristique `FirstRelease` est un attribut de type entier représentant l'année de première mise à disposition du logiciel. Les valeurs sont extraites de la MCP de la Figure 5.10. La date pour le logiciel `ClearCase` n'étant pas donnée de manière précise, la valeur par défaut "*" est assignée. Cette caractéristique donne une seule valeur pour chaque produit. Les caractéristiques `ClientServer`, `Distributed`, `Merge` et `Lock` sont des *features*. Elles proviennent des deux caractéristiques `Repository Model` et `Concurrency Model` de la MCP de la Figure 5.9 sur lesquelles on a appliqué un *scaling* binaire. Nous avons vu précédemment que les attributs étaient un moyen de définir de grands domaines de valeurs sans complexifier le modèle de variabilité. Or, le domaine de valeurs de ces deux caractéristiques étant très peu étendu (deux valeurs possibles et non booléennes) nous avons considéré dans cas-là qu'elles pouvaient être représentées par des *features*. La

caractéristique `ProgrammingLanguage` est un attribut dont les valeurs sont des littéraux. Contrairement aux deux caractéristiques précédentes, son domaine de valeurs est plus étendu et n'est donc pas transformé en *features*. Chaque produit peut avoir plusieurs valeurs pour cette caractéristique, qui sont séparées par des points-virgule.

TABLE 5.2 – Extrait des MCPs des Figures 5.9 et 5.10 mises au bon format

Software	ClientServer	Distributed	Merge	Lock	ProgrammingLanguage	FirstRelease
Git		×	×		C ; shell scripts ; perl	2005
CVS	×		×		C	1986
ClearCase	×		×	×	C ; java ; perl	*
GnuArch		×	×		C ; shell scripts	2005
CVSNT	×		×	×	C++	1998

5.4.3 Des matrices de comparaison de produits aux vecteurs de patterns

Afin de traiter leurs valeurs dans le cadre des structures de patterns, nous représentons chaque produit de la MCP par un vecteur de patterns correspondant à sa description. Si chaque pattern composant ces vecteurs appartient à une taxonomie, on peut construire la taxonomie associée aux vecteurs de patterns automatiquement, tel que défini en Section 5.3. Nous serons ensuite capable d'y appliquer l'AFC et les structures de patterns pour extraire des informations sur la variabilité. Nous présentons donc comment transformer une MCP bien formée en une taxonomie de vecteurs de patterns notée $(\mathcal{D}_v, \sqcap_v)$.

Une MCP représente de manière tabulaire une collection de valeurs de caractéristiques pour chaque produit qu'elle documente. Intuitivement, nous pouvons déjà faire le rapprochement avec des vecteurs, combinant chacune des valeurs pour un produit. Cependant, pour pouvoir y appliquer l'AFC et les structures de patterns, l'ensemble des valeurs d'une caractéristique (i.e., ayant le même indice dans les vecteurs) doit pouvoir être organisé par spécialisation dans une taxonomie. Les valeurs que l'on trouve dans une MCP peuvent correspondre à deux types de données : les *features* (caractéristiques booléennes) et les attributs multivalués (caractéristiques multivaluées). Nous allons étudier ces deux types de données et définir comment les intégrer dans les vecteurs de patterns. Les *features* peuvent bien évidemment être vues comme des attributs ayant deux valeurs possibles (vrai/faux). Nous traitons cependant ce cas à part pour deux raisons. La première est que les *features* ont une importance et une signification bien différentes des attributs dans la modélisation des LPLs. Il est donc important de faire la distinction entre ces deux types de données. La seconde est que le caractère booléen des *features* permet un traitement plus efficace et centralisé que les autres attributs, comme nous le détaillerons ci-après.

Traitement des caractéristiques booléennes représentant des *features*

Une *feature* apparaît dans les MCPs sous la forme d'une caractéristique ayant deux valeurs possibles, vrai ou faux, indiquant si le produit logiciel la possède ou non. Considérer que chaque caractéristique booléenne représente un pattern dans un vecteur demande de créer des taxonomies sur ce domaine de valeurs booléen. Cette solution, en plus de n'apporter aucune nouvelle information, complexifie à la fois les vecteurs de patterns en rajoutant des taxonomies de valeurs inutiles, ainsi que la structure de treillis finale.

Une solution plus efficace consiste à considérer que l'ensemble des caractéristiques booléennes d'une MCP constitue un seul type de pattern. Dans ce cas-là, on considère la partie booléenne de la MCP, i.e., l'ensemble des caractéristiques {vrai, faux}, comme étant un contexte formel. Dans notre exemple, cela correspond aux 4 premières colonnes de la

Table 5.2. On applique ensuite l'AFC traditionnelle sur ce contexte formel pour en extraire une taxonomie de valeurs. En effet, nous avons vu précédemment que l'AFC était un sous-cas des structures de patterns dans lequel les patterns étaient des ensembles d'attributs booléens et l'opérateur de similarité était défini par l'intersection ensembliste $\cap$. À partir d'ensembles d'attributs binaires, l'AFC permet de construire un unique treillis de concepts dans lequel les attributs sont organisés par spécialisation. Cette taxonomie structure des sous-ensembles d'attributs booléens en fonction de sous-ensembles plus généraux, i.e., contenant moins d'attributs booléens. Cette solution a l'avantage de 1) alléger les vecteurs et la structure finale en considérant les *features* comme un seul ensemble de patterns et 2) créer automatiquement une taxonomie (unique) par l'intermédiaire de l'AFC et des treillis de concepts.

La Figure 5.11 (gauche) représente le treillis de concepts associé au contexte formel formé par les 4 quatre premières colonnes de la Table 5.2. La Figure 5.11 (droite) présente la taxonomie extraite de ce treillis. On peut y lire que tous les logiciels ayant la *feature* `Lock` ont aussi la *feature* `ClientServer`, ou que tous les logiciels ont la *feature* `Merge`.

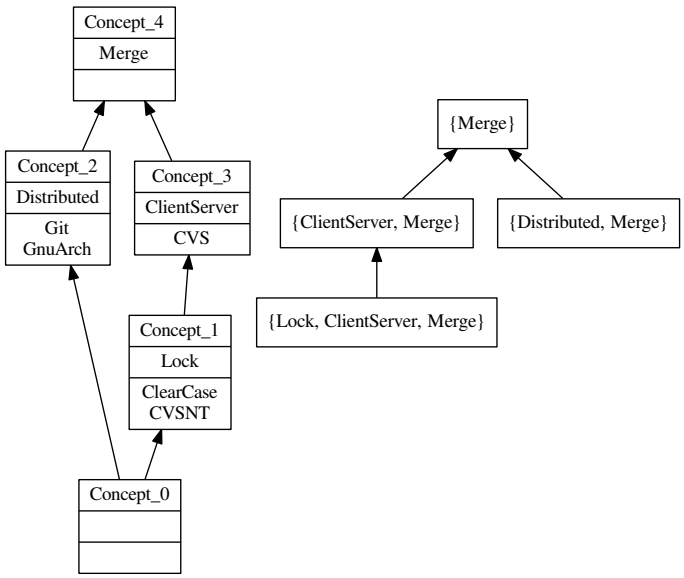


FIGURE 5.11 – (gauche) Treillis des concepts obtenu à partir de l'ensemble des caractéristiques booléennes de la MCP de la Table 5.2 et (droite) taxonomie induite par le treillis des concepts

Par conséquent, pour chaque MCP, l'ensemble des *features* sera représenté par un ensemble de patterns que l'on notera $\mathcal{D}_f$. L'opérateur de similarité équivalent à l'intersection ensembliste permet de plus de définir la taxonomie $(\mathcal{D}_f, \cap)$ de manière automatique.

Traitement des caractéristiques multivaluées représentant des attributs

Nous avons vu précédemment que pour que les valeurs $a_1, ..., a_n$ d'un attribut a puissent être utilisées dans un vecteur de patterns, elles doivent être organisées par spécialisation. C'est-à-dire que toutes les valeurs de a doivent représenter un pattern dans $\mathcal{D}_a$ et qu'il doit exister (au moins) un opérateur de similarité $\sqcap_a$ sur $\mathcal{D}_a$. Il est peu probable que l'ensemble des valeurs de a contenues dans la MCP représentent la totalité des valeurs nécessaires pour définir la taxonomie $(\mathcal{D}_a, \sqcap_a)$, comme illustré précédemment dans la Figure 5.6. De ce fait, il est nécessaire de compléter la taxonomie avec les valeurs manquantes.

◦ *Opérateur de similarité sur les nombres :* Selon le type des valeurs de l'attribut et l'opérateur de similarité défini, cette complétion peut se faire de manière automatique, ou manuelle. En effet, s'il existe une formule qui permet de calculer la similarité de deux valeurs de $(\mathcal{D}_a, \sqcap_a)$, la construction de la taxonomie à partir d'un ensemble de valeurs de départ peut se faire de manière automatique. C'est le cas par exemple avec l'opérateur de similarité $\sqcap_{inter}$ sur les intervalles de nombres entiers, tel que présenté dans la Section 5.3. Les attributs représentant des entiers ou des réels sont de bons candidats pour la construction automatique de taxonomies. Il est possible de définir un opérateur de similarité (associé à une formule) par défaut, ou d'en proposer plusieurs à l'expert du domaine. Par exemple, la formule $a_1 \sqcap_{\leq} a_2 = max(a_1, a_2)$ est une autre façon de définir automatiquement une taxonomie sur des nombres. La Figure 5.12 présente les taxonomies construites automatiquement sur les valeurs de l'attribut `FirstRelease` de la Table 5.2 avec (à gauche) l'opérateur de similarité construisant des intervalles $\sqcap_{inter}$ et (à droite) l'opérateur de similarité $\sqcap_{\leq}$. Il est aussi possible de laisser l'expert définir manuellement la taxonomie.

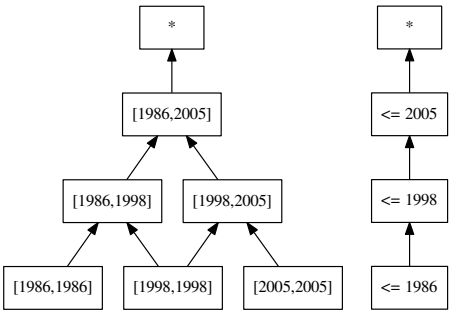


FIGURE 5.12 – Deux taxonomies construites automatiquement sur les valeurs de l'attribut `FirstRelease` de la Table 5.2

◦ *Opérateur de similarité sur les littéraux :* Lorsque les valeurs d'un attribut sont des littéraux, il est plus difficile de compléter automatiquement la taxonomie. Tout d'abord, une taxonomie extérieure peut exister, provenant d'une ontologie métier telles que `DBPedia` [4] ou `BioPortal` [5]. Si on n'en dispose pas, on peut proposer une taxonomie construite automatiquement. Pour ce type d'attribut, plusieurs cas sont possibles. Nous avons vu précédemment qu'un attribut d'une MCP peut donner plusieurs valeurs pour un produit (i.e., attribut multivalué à valeurs multiples). C'est le cas par exemple de l'attribut `ProgrammingLanguage`. Si certaines valeurs sont partagées par plusieurs produits (i.e., la valeur *C* ou *shell scripts* dans notre exemple), on peut utiliser l'AFC pour construire automatiquement une taxonomie basée sur l'intersection ensembliste des ensembles de littéraux. Dans ce cas-là, un pattern de la taxonomie ne représente pas une des valeurs de l'attribut (i.e., un littéral), mais un ensemble de littéraux. Pour construire cette taxonomie, on considère que chaque littéral est un attribut booléen de l'AFC et on construit le contexte correspondant. Pour l'attribut `ProgrammingLanguage`, on obtient le contexte formel de la Figure 5.13 (gauche). Le treillis de concepts correspondant et la taxonomie qui en est extraite sont présentés dans la Figure 5.13 (centre, droite).

Dans le cas où les valeurs de l'attribut a sont uniques pour chaque produit (i.e., attribut multivalué à valeurs uniques), l'AFC ne groupe pas de littéraux et on obtient une taxonomie spécifiant que toutes les valeurs de a sont incomparables, en utilisant la valeur de dissimilarité `"*"`. Cependant, même si cette solution permet à l'attribut d'être traité par le cadre

4. https ://wiki.dbpedia.org/
5. https ://bioportal.bioontology.org/

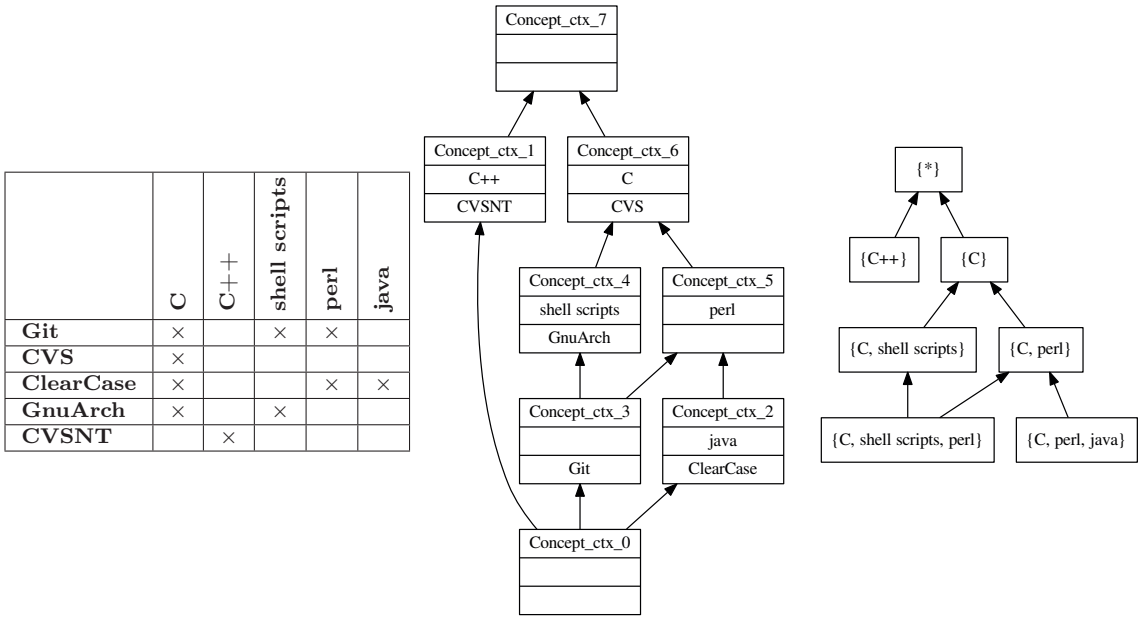


FIGURE 5.13 – Construction automatique de taxonomie pour les attributs ayant plus d'un littéral par produit : (gauche) contexte formel correspondant aux valeurs de l'attribut `ProgrammingLanguage`, (centre) treillis de concepts associé, (droite) taxonomie extraite

mathématique, on perd l'intérêt de l'utilisation des structures de patterns. La Figure 5.14 présente la taxonomie spécifiant que les valeurs de l'attribut `ProgrammingLanguage` sont toutes incomparables, aussi appelé *nominal scaling*. Pourquoi garde-t-on dans ce cas-là un ensemble de littéraux comme pattern et non pas un littéral ? Dans le cas où l'attribut peut avoir plusieurs valeurs pour un produit, il faudrait donner la similarité dans chaque paire de ces valeurs dans la description du produit. Or, toutes les valeurs étant incomparables, chaque produit présentant plusieurs valeurs pour un attribut possèderait alors la valeur de dissimilarité `"*"`.

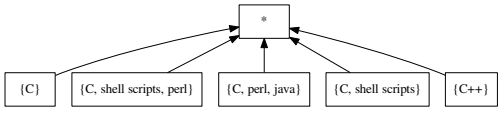


FIGURE 5.14 – Exemple de taxonomie spécifiant que les valeurs de l'attribut `ProgrammingLanguage` sont incomparables entre elles

Lorsqu'un attribut donne au plus un seul littéral par produit, on retombe dans le cas où les valeurs sont incomparables, mais cette fois-ci un pattern représente un seul littéral.

Dans tous les cas, si une taxonomie extérieure n'est pas utilisée, l'expert pourra la définir manuellement.

◦ *Influence du choix de l'opérateur de similarité :* Il est important de noter que le choix de l'opérateur de similarité ici peut influencer 1) le nombre de relations extraites impliquant des valeurs d'attributs et 2) la forme des relations extraites :

1. En effet, le choix de l'opérateur de similarité peut influencer le nombre de généralisations dans la taxonomie (voir Figure 5.12 et Figures 5.14 et 5.13). Plus ce nombre

est petit, plus le nombre de relations pouvant être extraites est susceptible d'être réduit. Le nombre de valeurs dans la taxonomie est cependant borné : soit n le nombre de valeurs de a dans la MCP, m le nombre de groupes disjoints de valeurs apparaissant ensemble avec $m \leq n$, la taxonomie possède au moins $m+1$ valeurs (tous les groupes de valeurs + "*", cas de la Figure 5.14) et au plus 2^n valeurs. Si l'expert souhaite réduire le nombre de relations extraites, cette piste peut être étudiée.

2. La façon dont sont calculées et représentées les généralisations dans la taxonomie va influencer la façon dont ses valeurs apparaissent dans les relations extraites. Par exemple, l'utilisation de l'opérateur de similarité $\sqcap_{inter}$ extraira des relations incluant des valeurs de la forme $att : [a_i, a_j]$. Si l'opérateur de similarité utilisé est $\sqcap_{\leq}$, les valeurs impliquées dans les relations seront de la forme $att : \leq a_i$. L'expert peut donc choisir à travers l'opérateur de similarité la forme des valeurs incluses dans les relations extraites.

◦ *De l'importance des taxonomies :* Si plusieurs relations de même type concernent plusieurs valeurs d'un attribut, il est possible de réduire le nombre de ces relations en prenant en compte la généralisation de ces valeurs telle que donnée par l'opérateur de similarité et la taxonomie. Nous avons vu que pour pouvoir traiter un attribut multivalué avec ce cadre mathématique, réaliser une taxonomie spécifiant que toutes ses valeurs sont incomparables, à la manière de la Figure 5.14, est suffisant. Seulement, si aucune abstraction / généralisation n'est introduite dans la taxonomie, il ne sera pas possible de réduire le nombre de relations, d'où l'importance d'avoir des taxonomies plus complexes. Nous étudions la réduction des relations logiques extraites grâce aux taxonomies de valeurs dans la section suivante.

Assemblage des vecteurs de patterns

Une fois définies les taxonomies associées aux attributs multivalués, ainsi que celle correspondant aux *features*, il est possible de composer les vecteurs qui seront organisés dans $(\mathcal{D}_v, \sqcap_v)$.

Définition 5.5 (Vecteur de patterns associé à une MCP)**.** *Soient $A = \{a_1, a_2, \ldots, a_n\}$ l'ensemble des attributs multivalués d'une MCP et $(\mathcal{D}_1, \sqcap_1), (\mathcal{D}_2, \sqcap_2), \ldots, (\mathcal{D}_n, \sqcap_n)$ les taxonomies qui leur sont associées. Soient F l'ensemble des features de la MCP et $(\mathcal{D}_f, \cap)$ la taxonomie qui lui est associée. Chaque produit de la MCP peut être décrit par un vecteur de la forme :*

$$\langle d_1, d_2, \ldots, d_n, d_f \rangle, \text{ avec } d_i \in (\mathcal{D}_i, \sqcap_i), \forall i \in \{1, 2, \ldots, n\} \text{ et } d_f \in (\mathcal{D}_f, \cap)$$

La MCP de la Table 5.2 possède deux attributs et quatre *features*. Un vecteur représentant une description de cette MCP possédera donc trois éléments : un pour chaque attribut et un pour l'ensemble de *features*. Supposons que l'on choisisse l'opérateur de similarité $\sqcap_{inter}$ pour l'attribut `FirstRelease` (Fr) (i.e., taxonomie $(D_{fr}, \sqcap_{inter})$ de la Figure 5.12 (gauche)) et l'opérateur de similarité $\cap$ pour l'attribut `ProgrammingLanguage` (Pl), correspondant à la taxonomie $(D_{pl}, \cap)$ de la Figure 5.13. On définit alors les vecteurs de patterns représentant la description de chaque produit comme suit :

$$\begin{aligned}
\text{Git} &= \langle Fr = 2005, Pl = \{C, perl, shell\ scripts\}, \{Distributed, Merge\} \rangle \\
\text{CVS} &= \langle Fr = 1986, Pl = \{C\}, \{ClientServer, Merge\} \rangle \\
\text{ClearCase} &= \langle Fr = *, Pl = \{C, java, perl\}, \{ClientServer, Merge, Lock\} \rangle \\
\text{GnuArch} &= \langle Fr = 2005, Pl = \{C, shell\ script\}, \{Distributed, Merge\} \rangle \\
\text{CVSNT} &= \langle Fr = 1998, Pl = \{C++\}, \{ClientServer, Merge, Lock\} \rangle
\end{aligned}$$

On peut alors déduire la similarité entre deux vecteurs de manière automatique :

$$\begin{gathered}\text{Git} \sqcap_{vp} \text{CVS} \\ = \langle Fr = 2005 \sqcap_{inter} 1986, Pl = \{C, perl, shell\ scripts\} \sqcap_{pl} \{C\}, \\ \{Distributed, Merge\} \cap \{ClientServer, Merge\}\rangle \\ = \langle Fr = [1986, 2005], Pl = \{C\}, \{Merge\}\rangle\end{gathered}$$

La taxonomie de vecteurs de patterns associée peut donc aussi être construite automatiquement. Notons que la taxonomie de vecteurs de patterns associée à une MCP présente les mêmes informations que la MCP initiale, plus d'autres concernant la similarité des valeurs de ces caractéristiques. La description représentée par une MCP est donc incluse dans la description représentant sa taxonomie de vecteurs de patterns associée.

En résumé (5.4) :

- Les **matrices de comparaison de produits** représentent des descriptions de produits logiciels complexes car elles possèdent des caractéristiques booléennes et multivaluées
- Ces matrices doivent être nettoyées pour respecter un bon format afin d'être traitées correctement par un processus automatique.
- Chaque **caractéristique multivaluée** représente un **attribut** dont les valeurs doivent faire l'objet d'une **taxonomie (demi-treillis supérieur)** construite à l'aide d'un **opérateur de similarité**. Nous avons identifié des cas où ces taxonomies peuvent être construites automatiquement.
- Chaque **caractéristique booléenne** représente une ***feature***. L'ensemble des *features* fait l'objet d'**une seule taxonomie**, qui peut être construite automatiquement à l'aide de l'**AFC**.
- Une fois l'ensemble des taxonomies définies, chaque produit de la matrice de comparaison peut être décrit par un **vecteur de patterns**. Un vecteur de patterns possède un pattern par attribut et un pattern pour l'ensemble des *features*.
- La définition de ces taxonomies sera utile pour **réduire le nombre de relations logiques représentant la variabilité étendue**.
- La description représentée par une MCP est incluse dans la description représentant sa taxonomie de vecteurs de patterns associée.

5.5 Sur l'extraction de variabilité étendue

Dans cette section, nous nous intéressons dans un premier temps à la construction du treillis des concepts de patterns à partir de la structure de vecteurs de patterns obtenue à l'étape précédente. Dans un second temps, nous étudions l'extraction depuis ce treillis des relations logiques représentant la variabilité de ces descriptions multivaluées. Les relations que l'on cherche à extraire sont celles identifiées dans la Section 5.2 : les cooccurrences, implications et mutex complexes, i.e., impliquant au moins une valeur d'attribut. Nous montrons comment utiliser les taxonomies pour réduire le nombre de relations extraites sans perte d'information. Par la suite, nous qualifions cette étape de réduction d'*élimination de la redondance*, car nous omettons certaines relations qui peuvent être inférées grâce aux hiérarchies de spécialisation/généralisation des taxonomies de valeurs. Les groupes de *features* (au sens de groupes de caractéristiques des FMs) étant, comme leur nom l'indique, constitués uniquement de *features*, ils ne sont pas étudiés ici. Cependant, leur processus d'extraction ne diffère pas de celui déjà vu précédemment et peut être appliqué de la

même manière sur les treillis de concepts de patterns. Enfin, nous étudions une autre méthode de réduction des relations extraites. Alors que l'élimination de la redondance est une réduction d'ordre logique (i.e., les relations omises peuvent être inférées à partir des relations gardées), cette autre méthode de réduction est plutôt d'ordre ontologique, car elle cherche à ne garder que les relations entre paires d'éléments dont les interactions intéresse un expert.

Cette section correspond à la phase de fouille de données dans le processus d'extraction de connaissances.

5.5.1 Construire le treillis des concepts de patterns

Nous avons vu dans la Section 5.3 les éléments théoriques de la construction d'un treillis de concepts à partir d'une taxonomie de vecteurs de patterns. Cependant, il n'existe à ce jour aucun outil public permettant de construire un treillis à partir d'une structure de vecteurs de patterns aussi complet que ceux de l'AFC traditionnelle.

Scaling binaire

Il est possible de formater les taxonomies de vecteurs de patterns afin d'utiliser les outils définis pour l'AFC traditionnelle, grâce à la technique de *scaling* binaire. Le *scaling* binaire consiste généralement en la transformation de matrices multivaluées en contextes formels (matrice booléenne) dans le but d'être traitées par l'AFC. Pour cela, les caractéristiques multivaluées sont "échelonnées", c'est-à-dire que leurs valeurs sont transformées en des attributs booléens, formant ainsi un contexte formel. Nous donnons ci-après une définition du *scaling* binaire basique. Il existe d'autres types de *scaling* binaires plus complexes [GW99].

Définition 5.6 (*scaling* binaire (basique) d'une matrice multivaluée)**.** *Soit une matrice multivaluée $\mathcal{M}$ possédant n caractéristiques multivaluées $C = \{c_1, c_2, \ldots, c_n\}$, un ensemble de produits $P = \{p_1, p_2, \ldots, p_m\}$ et définissant une fonction $Dom(c), c \in C$ donnant l'ensemble des valeurs de la caractéristique c et une fonction $P \times C \mapsto \bigcup_{i=1}^{n} Dom(c_i)$, avec pour $p \in P, c \in C$, $\gamma(p, c)$ donne les valeurs de la caractéristique c pour le produit p.*

Le scaling binaire basique de $\mathcal{M}$ produit un contexte formel $K = (O, A, I)$ tel que :

- $O = P$
- $A = \bigcup_{i=1}^{n} Dom(c_i)$
- $I = \{(p, a) | \exists c \in C \text{ et } a \in \gamma(p, c)\}$

Le contexte formel obtenu possède donc, en plus des caractéristiques booléennes de la MCP de départ, autant d'attributs booléens que de valeurs différentes d'attributs multivalués.

Dans notre cas, nous avons une structure de vecteurs de patterns et non plus une matrice multivaluée. Nous définissons alors le *scaling* binaire pour les taxonomies de vecteurs de patterns, où les valeurs de chaque taxonomie dont les éléments composent les vecteurs sont transformées en attribut booléen. Ce *scaling* est basé sur le *scaling* interordinal définit dans [GW99].

Définition 5.7 (*scaling* binaire d'une structure de vecteurs de patterns)**.** *Soit une structure de vecteurs de patterns $P_s = (R, (\mathcal{D}_{vp}, \sqcap_{vp}), \delta)$, où les vecteurs de $(\mathcal{D}_{vp}, \sqcap_{vp})$ sont composés de n éléments appartenant aux taxonomies $\{(\mathcal{D}_1, \sqcap_1), (\mathcal{D}_2, \sqcap_2), \ldots, (\mathcal{D}_n, \sqcap_n)\}$.*

Le scaling binaire de P_s produit un contexte formel $K = (O, A, I)$ tel que :

- $O = R$

— $A = \bigcup_{i=1}^{n} \mathcal{D}_i$
— $I = \{(o, d_1) | \exists d_2 \in \delta(o), d_1, d_2 \in \mathcal{D}_i, i \in \{1, 2, \ldots, n\},\ d_1 \sqsubseteq_i d_2\}$

Cette fois, le contexte formel obtenu possède autant d'attributs booléens que d'éléments dans l'ensemble des taxonomies. Appliqué à notre structure de vecteurs de patterns obtenue à la fin de la Section 5.4, le *scaling* binaire produit le contexte formel de la Table 5.3.

TABLE 5.3 – Contexte formel après *scaling* binaire de la structure de vecteurs de patterns

ctx	**ClientServer**	**Distributed**	**Merge**	**Lock**	**Fr :1986**	**Fr :1998**	**Fr :2005**	***Fr :[1998-2005]***	***Fr :[1986-1998]***	***Fr :[1986-2005]***	***Fr :****	**Pl :C**	**Pl :shell scripts**	**Pl :perl**	**Pl :java**	**Pl :C++**	***Pl :****
Git		×	×				×	×		×		×	×	×			
CVS	×		×		×				×	×		×					
ClearCase	×		×	×							×	×		×	×		
GnuArch		×	×				×	×		×		×	×				
CVSNT	×		×	×		×		×	×	×						×	

Le *scaling* binaire d'une structure de vecteurs de patterns construite à partir d'une matrice multivaluée et le *scaling* binaire basique de cette matrice multivaluée ne produisent donc pas le même contexte formel. Plus précisément, le *scaling* de la structure de patterns produit un contexte formel ayant les mêmes attributs booléens que le *scaling* de la matrice, ainsi que d'autres attributs booléens correspondant aux valeurs de similarité (ou généralisation) ajoutées dans les taxonomies d'éléments constituant les vecteurs. Dans notre exemple, le *scaling* binaire de la matrice de la Table 5.2 produit le contexte formel de la Table 5.3 sans les attributs en italique. Le treillis des concepts de la Table 5.3 est présenté dans la Figure 5.15. Ce treillis contient les mêmes informations qu'un treillis des concepts de patterns. Le *scaling* de la taxonomie de vecteurs de patterns produit donc plus d'attributs booléens que le *scaling* simple de sa MCP associée. Contrairement au *scaling* simple, il introduit des groupes / généralisations de valeurs des différents caractéristiques de la MCP et ce en plus des valeurs de départ. De la même façon que la MCP est incluse dans sa taxonomie de vecteurs de patterns associée, l'ensemble des attributs issu du *scaling* binaire (simple) de la MCP est inclus dans l'ensemble des attributs issu du *scaling* binaire de sa taxonomie de vecteurs de patterns associée. Le contexte formel associé à la taxonomie de vecteurs de patterns issue d'une MCP devrait permettre d'extraire un nombre de relations logiques plus élevé que le contexte formel issu du *scaling* simple de la MCP. Cependant, nous pensons que, après élimination de la redondance, le contexte formel contenant des attributs booléens représentant des groupes / généralisations de valeurs permet d'extraire moins de relations.

Lecture du treillis obtenu

On retrouve dans la structure de ce treillis les taxonomies de chaque attribut de nos vecteurs de patterns. Par exemple, la Figure 5.16 met en évidence la taxonomie associée à l'attribut `FirstRelease`. Le treillis de concepts de patterns est donc une représentation canonique d'une combinaison de taxonomies.

Il est aussi possible de reconstituer les vecteurs de patterns représentant les descriptions de chaque produit. Un concept introduisant un produit (par exemple, le concept `Concept_3`

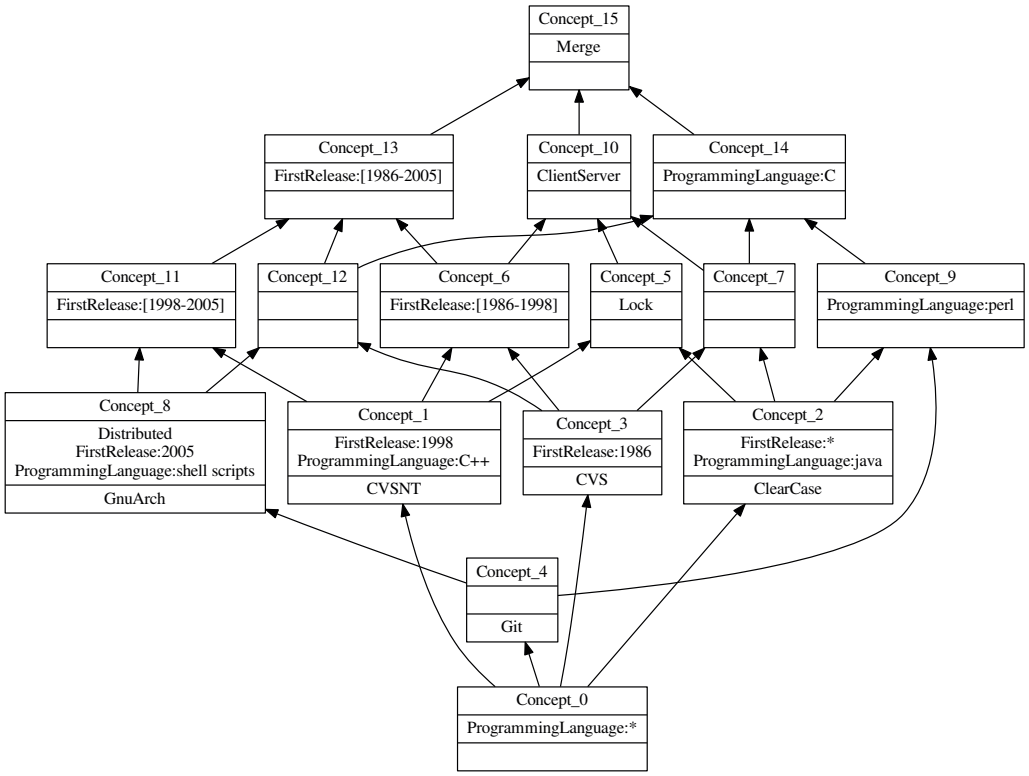


FIGURE 5.15 – Treillis des concepts de patterns de la Table 5.3

introduisant le produit *CVS*) possède dans son intension l'ensemble des patterns correspondant à ce produit. On retrouve l'ensemble des *features* (`ClientServer` et `Merge`), les patterns correspondant aux attributs (`ProgrammingLanguage:C` et `FirstRelease:1986`) ainsi que leur généralisation dans la taxonomie correspondante (`FirstRelease:[1986-1998]` et `FirstRelease:[1986-2005]` pour l'attribut `FirstRelease`). Ces valeurs sont représentées en vert dans la Figure 5.17. Afin de reconstituer le vecteur de patterns, il faut conserver l'ensemble des *features* et le pattern le plus spécialisé pour chaque attribut. Ces valeurs sont représentées en gras dans la Figure 5.17.

Le treillis de concepts ainsi obtenu depuis la structure de vecteurs de patterns met en évidence les relations logiques entre les éléments (*features* et valeurs d'attributs) correspondant à la variabilité expressive. Les méthodes d'extraction présentées ci-après sont *correctes* (toutes les relations extraites sont vraies pour l'ensemble de produits étudiés) et *complètes* (toutes les relations qui sont vraies pour l'ensemble de produits étudiés sont extraites).

5.5.2 Extraire les cooccurrences

Les éléments cooccurrents, i.e., apparaissant toujours ensemble dans les descriptions de produits, sont introduits dans l'intension du même concept. Par exemple, le `Concept_8` introduit trois éléments : la *feature* `Distributed` et les deux valeurs d'attributs `FirstRelease:2005` et `ProgrammingLanguage:shell scripts`. Cela signifie que tous les produits ayant un de ces éléments possèdent obligatoirement les deux autres. On dénotera ces cooccurrences par des équivalences logiques entre les éléments de chaque paire, par exemple :

$$(1)\ \mathit{Distributed} \leftrightarrow \mathit{FirstRelease} : 2005$$
$$(2)\ \mathit{ProgrammingLanguage} : \mathit{shell\ scripts} \leftrightarrow \mathit{Distributed}$$

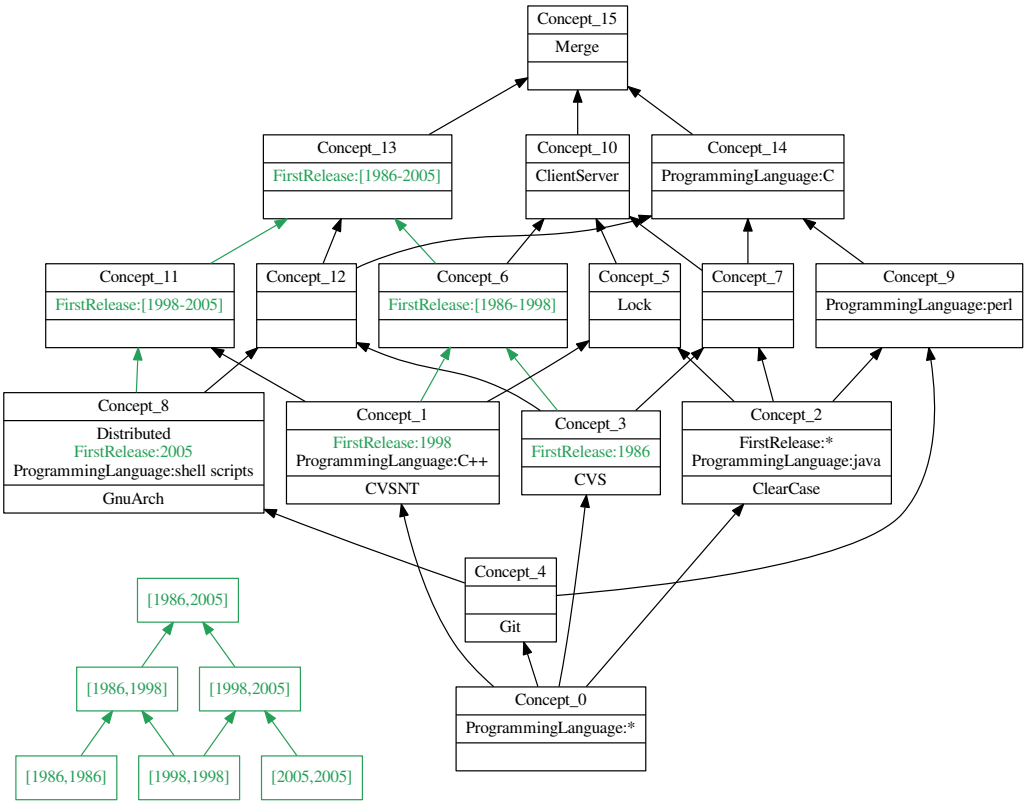


FIGURE 5.16 – Taxonomie des valeurs de l'attribut `FirstRelease` dans le treillis des concepts de patterns de la Table 5.3

(3) $FirstRelease : 2005 \leftrightarrow ProgrammingLanguage : shell\ scripts$

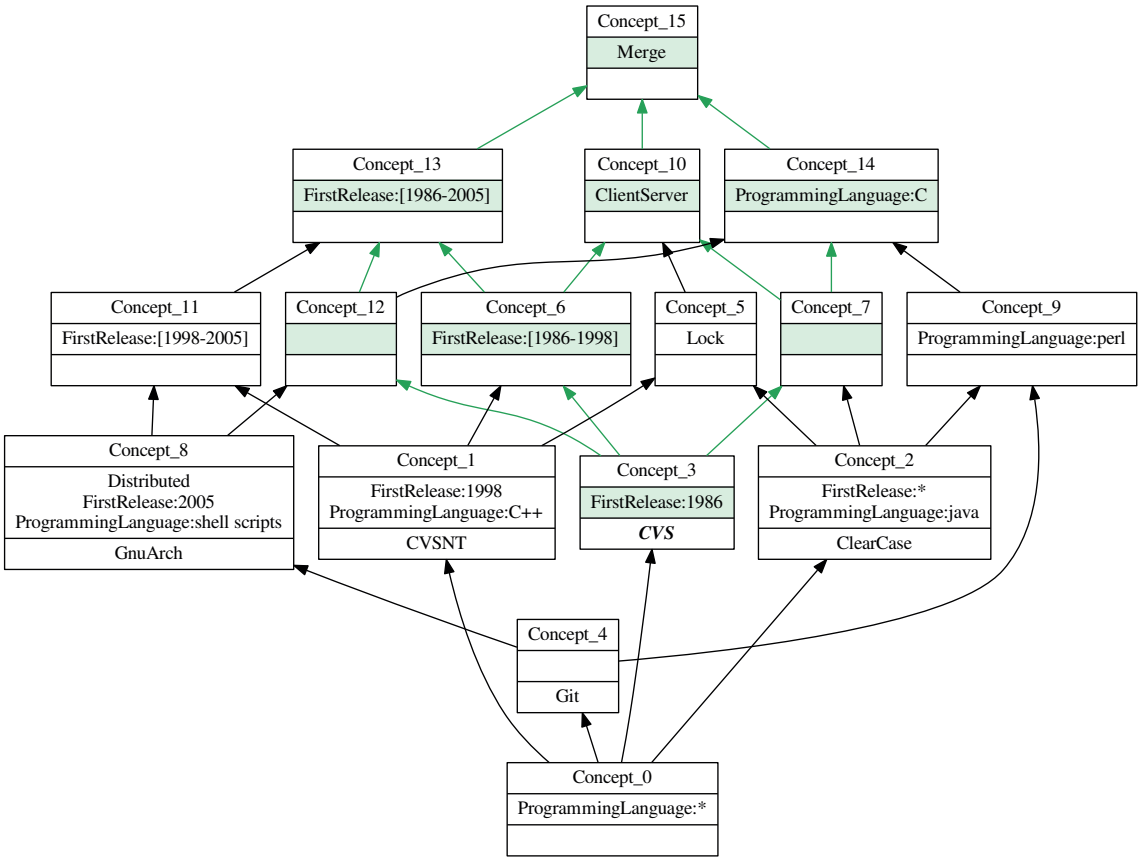
On peut ainsi extraire les cooccurrences entre *features* comme avec l'AFC traditionnelle, mais aussi des cooccurrences entre *features* et valeurs d'attributs (e.g., equivalences (1) et (2)) et entre valeurs d'attributs (e.g., équivalence (3)).

5.5.3 Extraire les implications binaires

Les implications binaires entre éléments peuvent être extraites en suivant la relation de spécialisation donnée par l'ordre $\leq_s$ entre les concepts. Cette relation est indiquée par les flèches entre les concepts dans la figure représentant le treillis. Puisqu'un concept hérite de tous les attributs booléens de ses super-concepts, un élément introduit dans un concept apparaîtra toujours avec les éléments introduits dans ses super-concepts dans la description d'un produit. En d'autres termes, si e_1 est introduit dans un sous-concept d'un concept introduisant e_2, alors tous les produits ayant e_1 dans leur description possèderont aussi e_2 : on le représente par une implication binaire de e_1 vers e_2 : $e_1 \rightarrow e_2$. Dans un treillis des concepts de patterns, on peut alors extraire des implications entre deux *features*, entre une *feature* et une valeur d'attribut et entre deux valeurs d'attributs.

Dans notre exemple, `Concept_2` $\leq_s$ `Concept_10` et ils introduisent respectivement la valeur d'attribut `ProgrammingLanguage:java` et la *feature* `ClientServer`. On a donc l'implication suivante :

$$ProgrammingLanguage : java \rightarrow ClientServer$$



$$\text{CVS} = \langle Fr = 1986, Pl = \{C\}, \{ClientServer, Merge\}\rangle$$

FIGURE 5.17 – Description du produit *CVS* dans le treillis des concepts de patterns de la Table 5.3

Conservation de la réduction transitive

Afin d'alléger le nombre d'implications extraites, nous pouvons conserver uniquement la réduction transitive de l'ensemble des implications, afin de construire le graphe d'implications binaires [ACP+12]. Cette extraction réduite peut se réaliser en choisissant d'extraire uniquement les implications entre les éléments introduits dans un concept (prémisse) et les éléments introduits dans ses super-concepts-attributs directs (conclusion). On appelle super-concepts-attributs directs les plus spécialisés des super-concepts introduisant un attribut. Les super-concepts-attributs directs du `Concept_3` sont les deux concepts `Concept_6` et `Concept_14`.

Dans notre exemple, l'implication $ProgrammingLanguage : java \rightarrow ClientServer$ ne serait donc pas explicitement extraite car `Concept_10` n'est pas un super-concept-attributs direct de `Concept_2`. Par contre, on extrait les implications suivantes :

$$ProgrammingLanguage : java \rightarrow Lock$$
$$Lock \rightarrow ClientServer$$

Ce qui permettra d'inférer, si besoin, l'implication $ProgrammingLanguage : java \rightarrow ClientServer$.

Cependant, il est à noter que la réduction transitive est utile lorsque la finalité de cette extraction est représentée sous forme de graphe d'implications binaires. Si les implications qui résultent de la réduction transitive ne sont pas analysées de manière globale (i.e.,

l'intégralité du graphe d'implications) mais de manière individuelle (chaque implication séparément), il est possible de perdre de l'information. Plus tard, nous analyserons la pertinence de chacune des implications extraites avec cette méthode ; dans ce cas-là, il faudra prendre en compte la fermeture transitive de ces implications et non pas sa réduction.

Élimination de la redondance

Il est possible dans certains cas de réduire le nombre d'implications grâce aux taxonomies. On étudie alors les implications dont au moins un des éléments représente une valeur d'attribut.

∘ *Implications binaires entre valeurs du même attribut :* Les implications entre deux valeurs du même attribut peuvent être omises, car ce type d'information se retrouve dans la taxonomie associée à l'attribut. Soit a_1 et a_2 deux valeurs de l'attribut a. Si $a_2 \sqsubseteq a_1$, alors tous les produits pouvant être caractérisés par la valeur a_1 sont aussi caractérisés par la valeur plus générale a_2. On a alors l'implication $a_1 \rightarrow a_2$. C'est le cas dans notre exemple pour l'implication $FirstRelease : 1986 \rightarrow FirstRelease : [1986 - 1998]$ (`Concept_3` $\leq_s$ `Concept_6`). Il peut être intéressant de les conserver pour les taxonomies de littéraux construites automatiquement (montrant l'émergence de dépendances entre les valeurs de l'attribut), mais moins nécessaire pour les attributs dont les valeurs sont numériques et dont la taxonomie est construite automatiquement à partir d'une formule.

∘ *Un élément impliquant des valeurs du même attribut :*
Soit e un élément (*feature* ou valeur d'attribut) et a_1, a_2 deux valeurs de l'attribut a :

$$e \rightarrow a_1$$
$$e \rightarrow a_2$$

Si $a_2 \sqsubseteq a_1$ dans la taxonomie, conserver uniquement l'implication $e \rightarrow a_1$ n'entraînerait aucune perte d'information car on pourrait inférer $e \rightarrow a_2$ grâce à l'implication $a_1 \rightarrow a_2$ issue de la taxonomie de l'attribut a. Plus généralement, si l'on a l'implication $e \rightarrow a_i$, alors tous les patterns plus généralisés que a_i apparaissent aussi en conclusion d'une implication dont la prémisse est e.

$$\text{Si } e \rightarrow a_i, \text{ alors } \forall a_j | a_j \sqsubseteq a_i, e \rightarrow a_j$$

On cherche donc, dans ce genre de cas, à ne conserver que les implications ayant les patterns les plus spécialisés en conclusion.

On notera que si l'élément e est en cooccurrence avec une troisième valeur a_3 de l'attribut a ($a_3 \leftrightarrow e$), on ne conserve aucune des deux implications. En effet, cela signifie que a_3 est introduit dans le même concept que e et donc dans un sous-concept des concepts introduisant a_1 et a_2. L'implication $e \rightarrow a_3$ est vraie et $a_1 \sqsubseteq a_3$, $a_2 \sqsubseteq a_3$. S'il n'y a pas de cooccurrence avec un pattern plus spécialisé et que a_1 et a_2 ne sont pas une spécialisation/généralisation l'un de l'autre, alors les deux implications sont conservées.

∘ *Plusieurs valeurs du même attribut impliquant le même élément :* Soit e un élément (*feature* ou valeur d'attribut) et a_1, a_2 deux valeurs de l'attribut a :

$$a_1 \rightarrow e$$
$$a_2 \rightarrow e$$

Si $a_2 \sqsubseteq a_1$, alors $a_1 \rightarrow a_2$. Cette fois-ci, c'est la seconde implication (i.e., ayant la valeur la plus générale) qu'il faut conserver et la première que l'on peut inférer. Plus généralement,

si l'on a l'implication $a_i \rightarrow e$, tous les éléments plus spécialisés que a_1 apparaissent aussi en prémisse d'une implication dont la conclusion est e.

$$\text{Si } a_i \rightarrow e, \text{ alors } \forall a_j | a_i \sqsubseteq a_j, a_j \rightarrow e$$

Il faut donc conserver les implications ayant en prémisse les valeurs les plus générales. Dans notre exemple, on peut extraire les implications suivantes :

$$FirstRelease : 1986 \rightarrow ClientServer$$
$$FirstRelease : 1998 \rightarrow ClientServer$$
$$FirstRelease[1986 - 1998] \rightarrow ClientServer$$

On peut conserver uniquement la troisième et inférer les deux premières car la taxonomie de la Figure 5.12 (gauche) spécifie que $FirstRelease : 1986 \rightarrow FirstRelease[1986 - 1998]$ et $FirstRelease : 1998 \rightarrow FirstRelease[1986 - 1998]$.

5.5.4 Extraire les mutex

Deux éléments sont considérés comme mutuellement exclusifs s'ils n'apparaissent jamais ensemble dans la description d'un même produit. Ces paires d'éléments sont alors introduites respectivement dans deux concepts dont le plus grand minorant dans le treillis correspond au concept le plus bas du treillis (*bottom concept* en anglais, dénoté $\perp$), à condition que le *bottom concept* ait une extension vide. Si ce concept possède une extension non vide, alors aucune paire d'éléments ne peut former de mutex. En effet, le *bottom-concept* hérite de tous les éléments du treillis et s'il possède au moins un produit, alors sa description comprend tous les éléments du treillis ; il n'y en a donc pas qui sont mutuellement exclusifs. De cette façon, on peut extraire des mutex entre deux *features*, une *feature* et une valeur d'attribut et deux valeurs d'attributs.

Élimination de la redondance

Afin d'alléger le nombre de relations extraites, on peut omettre les mutex entre valeurs du même attribut, si elles n'apportent pas d'informations importantes. Dans notre exemple, on peut extraire les deux mutex suivants :

$$(1)\ FirstRelease : 1986 \rightarrow \neg\ FirstRelease : 1998$$
$$(2)\ ProgrammingLanguage : shell\ scripts \rightarrow \neg\ ProgrammingLanguage : C++$$

Alors que le mutex (1) n'apporte aucune réelle information, l'exclusion entre l'utilisation des deux langages de programmation pourrait intéresser certains concepteurs.

Il n'est pas non plus nécessaire de conserver tous les mutex entre un élément e et plusieurs valeurs a_1, a_2 du même attribut a :

$$e \rightarrow \neg\ a_1$$
$$e \rightarrow \neg\ a_2$$

Considérons que $a_2 \sqsubseteq a_1$. Si deux concepts (introduisant respectivement e et a_2) ont pour plus grand minorant le *bottom-concept*, il en est de même pour leurs sous-concepts. Les valeurs d'attributs plus spécialisées (i.e., introduites dans les sous-concepts du concept introduisant a_2) sont donc toutes mutuellement exclusives avec l'élément e. Puisque $a_2 \sqsubseteq a_1$, a_1 est donc introduit dans un sous-concept de a_2 et $e \rightarrow \neg\ a_1$ peut être inférée. Par conséquent, nous devons conserver le pattern le plus général pour les mutex.

Par exemple, on peut extraire les deux mutex suivants :

(3) $Distributed \rightarrow \neg\ FirstRelease : 1986$
(4) $Distributed \rightarrow \neg\ FirstRelease : [1986 - 1998]$

On peut retrouver (3) à partir de (4), on ne conserve donc que la seconde.

Si a_1 ne généralise/spécialise pas a_2, alors on doit garder les deux mutex.

5.5.5 Conserver les interactions pertinentes entre les éléments

Nous définissons la pertinence d'une relation entre deux éléments comme suit : si l'on estime que les deux éléments ont une influence l'un sur l'autre, la relation représentant leur interaction est pertinente, sinon elle ne l'est pas.

Les descriptions que nous étudions (et les descriptions de FPLs en général) n'étant pas complètes (i.e., représentant tous les produits possibles) ou bien n'étant pas entièrement représentatives de la FPL, il est fortement probable que certaines relations puissent être vérifiées dans l'ensemble des produits de la description, mais pas du point de vue du domaine modélisé. Nous qualifions ces relations de *relations accidentelles*, car elles sont vraies dans l'ensemble considéré alors qu'elles ne le devraient pas.

Afin d'éliminer une partie de ces relations accidentelles de manière automatique, nous définissons un "graphe d'influence", dont les nœuds représentent les caractéristiques d'une MCP et les arêtes entre deux nœuds indiquent qu'ils ont une influence l'un sur l'autre. Si aucune arête n'est dessinée entre deux éléments, cela signifie qu'ils n'ont pas d'influence entre eux. Nous avons choisi de pondérer les arêtes pour distinguer deux cas : par 2 si l'on estime leur influence forte, par 1 si l'on estime qu'ils peuvent éventuellement avoir une influence entre eux.

La Figure 5.18 présente un graphe d'influence sur les caractéristiques de la MCP de la Table 5.2. On peut y lire que l'on estime que :

— la fonctionnalité `ClientServer` influence la fonctionnalité `Lock` et inversement ;
— la date de sortie (`FirstRelease`) peut influencer le choix du langage de programmation (`ProgrammingLaguage`) ;
— le choix du langage de programmation n'influence pas la fonctionnalité `Merge`.

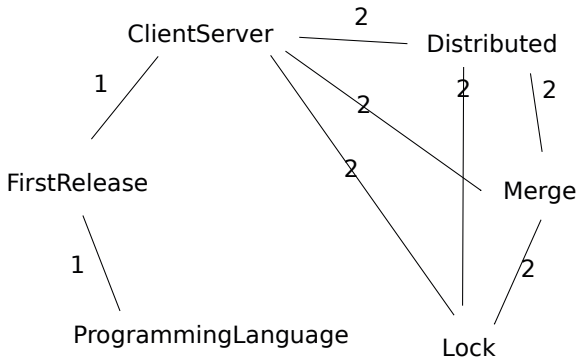


FIGURE 5.18 – Graphe d'influence sur les caractéristiques de la MCP de la Table 5.2

Les relations `Distributed` $\rightarrow \neg$ `Lock` et `Lock` $\rightarrow$ `ClientServer` que l'on peut extraire du treillis sont donc notées pertinentes. Les relations `FirstRelease:[1986-1998]` $\rightarrow$ `ClientServer` et `ProgrammingLanguage:C++` $\leftrightarrow$ `FirstRelease:[1998]` sont notées comme potentiellement pertinentes. Enfin, la relation `ProgrammingLanguage:Java` $\rightarrow$ `Lock` n'est pas prise en compte car jugée non pertinente et accidentelle.

Ce graphe peut définir les interactions pertinentes à extraire de manière générale dans une MCP, mais il peut aussi être construit pour une préoccupation donnée. Imaginons

qu'un expert ne s'intéresse qu'à l'ensemble des fonctionnalités des logiciels de la FPL (i.e., les caractéristiques `ClientServer`, `Distributed`, `Merge` et `Lock`) ; le graphe correspondant à cette préoccupation est présenté en Figure 5.19

De ce point de vue, les relations `FirstRelease:[1986-1998]` → `ClientServer` et `ProgrammingLanguage:C++` ↔ `FirstRelease:[1998]` sont cette fois notées comme non pertinentes.

En résumé (5.5) :

- On peut transformer une **structure de vecteurs de patterns** en un **contexte formel** équivalent, sans perte d'informations, grâce à un procédé appelé ***scaling*** *binaire*. On peut ainsi construire le treillis des concepts équivalent avec les outils de l'AFC traditionnelle.
- Le treillis de concepts de patterns est une **représentation canonique d'une combinaison de taxonomies**.
- On peut extraire des **cooccurrences**, des **implications** et des **mutex** contenant des *features* et/ou des valeurs d'attributs.
- Pour les **implications binaires**, on peut n'en conserver que la réduction transitive dans certains cas. On peut choisir d'ignorer les implications entre deux valeurs du même attribut. De plus, on peut ne garder que les implications ayant les valeurs les **plus spécialisées en conclusion** et les valeurs les **plus généralisées en prémisse**, sans perdre d'information.
- Pour les **mutex** contenant des valeurs d'attributs, on ne conserve que ceux ayant les **valeurs les plus générales**.
- On peut filtrer automatiquement les relations accidentelles en définissant les paires d'éléments qui ne s'influencent pas et dont les interactions sont jugées non pertinentes.

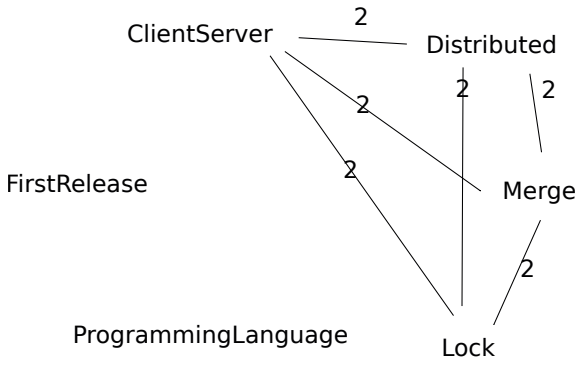


FIGURE 5.19 – Graphe d'influence sur les caractéristiques de la MCP de la Table 5.2 : préoccupation centrée sur les fonctionnalités des logiciels

5.6 Évaluation

Nous avons évalué la méthode exposée dans ce chapitre à travers quatre questions de recherche. La première concerne l'applicabilité de la méthode sur des descriptions de produits existantes. La deuxième et la troisième concerne le gain de la réduction des relations : dans un premier temps nous étudions le gain en nombre de relations de l'élimination de la

redondance, puis dans un second temps nous analysons le gain du filtrage des relations pertinentes avec des graphes d'influence. Enfin, la quatrième question analyse l'employabilité de la méthode après l'application des deux types de réduction.

RQ1 (applicabilité de la méthode) : Cette approche est-elle techniquement applicable sur des descriptions de produits du type MCP ? En effet, la taille des treillis de concepts est connue pour grandir de manière exponentielle en fonction des données d'entrées. De plus, dans la méthode proposée, les taxonomies mises en place pour pouvoir manipuler les FPLs comme des structures de patterns font encore grossir les données initiales. Pour répondre à cette question, nous nous intéressons à la taille des treillis des concepts de patterns générés par la méthode proposée à partir de matrices de comparaison de produits. Nous comparons aussi la taille des treillis avec celle des AOC-posets obtenus à partir des mêmes matrices, afin d'évaluer le gain de l'utilisation de ce sous-ordre.

RQ2 (gain de l'élimination de la redondance) : L'élimination de la redondance rendue possible grâce aux taxonomies permet-elle de réduire le nombre de relations extraites ? Nous avons vu précédemment que l'introduction de taxonomies dans les descriptions de produits augmentait le nombre de valeurs à prendre en compte et donc potentiellement le nombre de relations extraites. Nous allons dans un premier temps vérifier cette hypothèse et évaluer le taux d'augmentation du nombre de relations lorsque l'on utilise des taxonomies de valeurs pour traiter les caractéristiques des MCPs. Nous avons proposé des règles pour omettre certaines relations qui pouvaient être factorisées grâce à l'utilisation des taxonomies et ce sans perte d'information. Dans un second temps, nous allons évaluer le gain en nombre de relations extraites après application des règles d'élimination de la redondance. Nous espérons que, dans le cas général, l'introduction de taxonomies permettra d'obtenir un ensemble de relations plus restreint.

RQ3 (pertinence des relations extraites) : Les relations extraites sont-elles toutes pertinentes du point de vue du domaine ? Quel est le gain de leur élimination ? S'il existe des relations pertinentes, la méthode proposée permet de les extraire car elle est correcte et complète. Seulement, si beaucoup de relations sont vraies dans l'ensemble des données de départ mais non pertinentes du point de vue du domaine, elles seront extraites aussi. Les MCPs représentent un ensemble de produits qui ne sont potentiellement pas représentatifs de la FPL à laquelle ils appartiennent. De ce fait, certaines relations peuvent être valides dans la MCP car elle ne présente pas de produits prouvant leur invalidité. Nous cherchons ici à estimer le pourcentage de relations extraites qui sont pertinentes.

RQ4 (employabilité de la méthode) : Les relations extraites sont-elles nombreuses ? Les méthodes de réduction mises en place font-elles une grande différence ? Puisque la méthode proposée permet d'extraire toutes les implications / cooccurrences / exclusions mutuelles valides dans un ensemble de produits, leur nombre peut être potentiellement important, rendant difficile leur manipulation et leur compréhension. Nous cherchons donc à savoir à quel point le nombre de relations extraites peut être élevé et à quel point l'élimination de la redondance (définie pour les implications et les mutex dans la section précédente) ainsi que le filtrage par graphe d'influence réduit ce nombre.

5.6.1 Données et implémentation

Données d'entrée

Pour répondre aux questions de recherche précédentes, nous avons travaillé sur une trentaine de MCPs extraites de la catégorie *software comparisons* de Wikipedia[6]. Afin de récupérer les données de ces matrices, nous nous sommes appuyés sur l'outil `Open Compare`[7] qui permet d'extraire des matrices depuis Wikipedia et de les enregistrer au format `.csv`. Une base de données d'environ 1400 MCPs déjà extraites est aussi mise à disposition. Nous avons sélectionné des MPCs au hasard dans cette base, puis généré le fichier `.csv` correspondant. Les fichiers `.csv` obtenus représentent les données brutes telles qu'elles apparaissent dans la MCP originale. Il faut donc les nettoyer manuellement pour pouvoir les traiter par la suite avec notre méthode.

La Table 5.4 présente des informations sur les 26 MCPs sélectionnées et nettoyées. Elles sont disponibles dans le répertoire `data/clean_pcms/` du projet[8].

Table 5.4 – Chiffres sur les MCPs sélectionnées pour l'évaluation

	minimum	mean	maximum
#products	8	23	75
#boolean characteristics	0	5	17
#literal characteristics	0	2	4
#numerical characteristics	0	1	4
#cells	36	212	1650

La MCP avec le plus petit nombre de produits en possède 8 et la plus grande en possède 75. En moyenne, les MCPs étudiées présentent 23 produits. Certaines MCPs n'ont pas de caractéristiques booléennes, littérales, ni numériques. Les caractéristiques booléennes sont en moyenne plus nombreuses que les multivaluées. La plus petite MCP ne possède pas plus de 40 cellules, mais la plus grande en possède 1650. On peut noter que les MCPs sélectionnées pour conduire cette expérimentation peuvent être considérées comme des MCPs de taille importante. En effet, en se basant sur les résultats de Sannier et. al. [SAB13] qui ont réalisé une analyse quantitative sur plus de 165 matrices issues de Wikipedia, en moyenne, une MCP possède 178,5 cellules. La Table 5.4 montre que en moyenne, les MCPs sélectionnées sont plus grandes que la moyenne en terme de cellules.

Implémentation de la méthode proposée

Afin de répondre aux trois questions de recherche, nous avons mis en place une solution logicielle implémentant les étapes clés de la méthode proposée. Cette implémentation est disponible sur github.[9] Nous avons utilisé 7 fonctionnalités pour conduire cette évaluation.

1. Le programme permet de lire la MCP nettoyée au format `.csv` et d'afficher les valeurs distinctes de chaque caractéristique. Cela nous permet de vérifier si la mise au bon format a été correctement réalisée dans chaque colonne de la matrice.
2. Si la matrice est au bon format, le programme sépare automatiquement les caractéristiques booléennes (*features*) et les multivaluées (attributs). Il peut proposer à

6. https ://en.wikipedia.org/wiki/Category :Software_comparisons, dernier accès en mars 2018
7. https ://github.com/OpenCompare/OpenCompare
8. https ://github.com/jcarbonnel/CLEF
9. https ://github.com/jcarbonnel/CLEF

l'utilisateur de transformer les valeurs d'un attribut multivalué en *features* si cela lui semble pertinent. À la fin de cette étape, la composition des vecteurs de patterns est établie.

3. Le programme détecte le type de valeurs des attributs (littéraux, entiers ou réels) et crée automatiquement la taxonomie correspondante. Nous avons défini par défaut d'utiliser des intervalles pour les attributs numériques et l'AFC pour les littéraux. La MCP est maintenant sous la forme d'une structure de patterns.
4. Le programme réalise ensuite un *scaling* binaire complet de la structure de vecteurs de patterns. Le contexte formel ainsi obtenu est enregistré au format `.rcft`, qui peut être lu par le logiciel `RCAExplore` afin de construire le treillis et/ou l'AOC-poset correspondant.
5. Le logiciel appelle `RCAExplore` sur le fichier `.rcft` produit précédemment pour construire le treillis des concepts de patterns. Ce treillis est enregistré dans des fichiers au format `.dot` [EGK+01].
6. À partir des fichiers `.dot` représentant les concepts formels et l'ordre partiel, le logiciel extrait les mutex, implications et cooccurrences. Les relations sont filtrées pour enlever la redondance d'information et sont enregistrées dans des fichiers textes, triées en fonction de leur support.
7. Enfin, le programme peut trier les ensembles de relations extraites pour garder celles impliquant un ensemble d'éléments spécifiés.

5.6.2 Méthodologie

RQ1. (applicabilité de la méthode) Pour avoir un ordre de grandeur sur la taille des treillis de concepts et des AOC-posets obtenus avec notre méthode, nous utilisons les fonctionnalités 1, 2, 3, 4 et 5 de la solution logicielle mise en place. La première fonctionnalité nous sert à vérifier le nettoyage de la matrice. La deuxième fonctionnalité attend une décision de l'utilisateur, nous choisissons dans tous les cas de considérer les caractéristiques multivaluées comme des attributs et non des *features*. Les fonctionnalités 3, 4 et 5 se déroulent automatiquement ; une fois terminées, la MCP possède un dossier dans lequel on trouve le contexte formel équivalent ainsi que les fichiers `.dot` représentant les structures conceptuelles produites par `RCAExplore`. Nous analysons ces derniers : la taille d'une structure conceptuelle étant caractérisée par son nombre de concepts et la taille de l'ordre partiel (i.e., le nombre de flèches entre les concepts), nous relevons les deux types de données.

RQ2. (gain de l'élimination de la redondance) Pour répondre à cette question, nous générons 1) le nombre de relations extraites avec les taxonomies avant et après élimination de la redondance et 2) le nombre de relations extraites sans l'introduction des taxonomies. Pour connaître le nombre de relations extraites avant et après l'élimination de la redondance, nous utilisons la fonctionnalité 6 de la solution logicielle. Cette fonctionnalité permet d'enregistrer les trois types de relations étudiées dans des fichiers après avoir été filtrées. Nous avons cependant compté l'ensemble de relations trouvées dans les MCPs avant de les filtrer. Nous sommes donc capables de comparer le gain de l'élimination de la redondance pour les mutex et les implications. Pour le cas des implications, nous comparons le gain de la réduction transitive par rapport à la fermeture transitive, avant et après réduction de la redondance. Les cooccurrences n'ont pas de redondance, nous évaluons simplement leur nombre. Pour connaître le nombre de relations extraites sans utiliser de taxonomies, nous relançons les fonctionnalités 1, 2, 3, 4, 5 et 6 de la solution mise en place, sauf que cette fois, nous choisissons de considérer toutes les caractéristiques multivaluées

comme des *features*. Cela revient à réaliser un *scaling* binaire simple, tel que défini dans la Section 5.5.1 : aucune taxonomie n'est ainsi utilisée dans le processus d'extraction.

RQ3. (pertinence des relations extraites) Nous avons sélectionné 2 MCPs afin de les analyser de façon semi-automatique. Les critères de sélection sont les suivants : 1) la MCP doit posséder un nombre important de relations (au-dessus de la moyenne) mais sans qu'il soit trop difficile ni long à analyser manuellement et 2) les caractéristiques de la MCP doivent être toutes compréhensibles pour pouvoir estimer la pertinence des relations entre elles. Nous avons réalisé des "graphes d'influences" pour les deux MCPs sélectionnées ; ils sont présentés en Figures 5.20 et 5.21. Enfin, puisqu'il est possible que toutes les caractéristiques d'une matrice n'intéressent pas un expert, nous avons défini une à deux préoccupations par matrice et remanié le graphe d'influence correspondant à chacune de ces préoccupations. Pour chaque matrice, nous étudions donc deux types de graphes : un général et d'autres plus spécifiques en fonction des préoccupations. Nous évaluons ainsi le pourcentage de relations non pertinentes dans un cas général, ainsi que dans des cas plus spécifiques. Ce filtrage est appliqué sur l'ensemble des relations extraites après élimination de la redondance. Notons que, puisque nous étudions ici chaque implication séparément, nous travaillons donc sur la fermeture transitive.

Nous avons tout d'abord sélectionné la matrice `accounting_software_2`. Nous avons défini deux préoccupations : la première concerne "l'évolution" des logiciels de cette famille et s'intéresse aux dates de sorties des différentes versions et du statut de développement et la seconde concerne le point de vue "développement" et s'intéresse plutôt aux langages de programmation et à la base de données. Les trois graphes d'influence de cette matrice sont présentés dans la Figure 5.20. Ils ont été validés par 3 personnes.

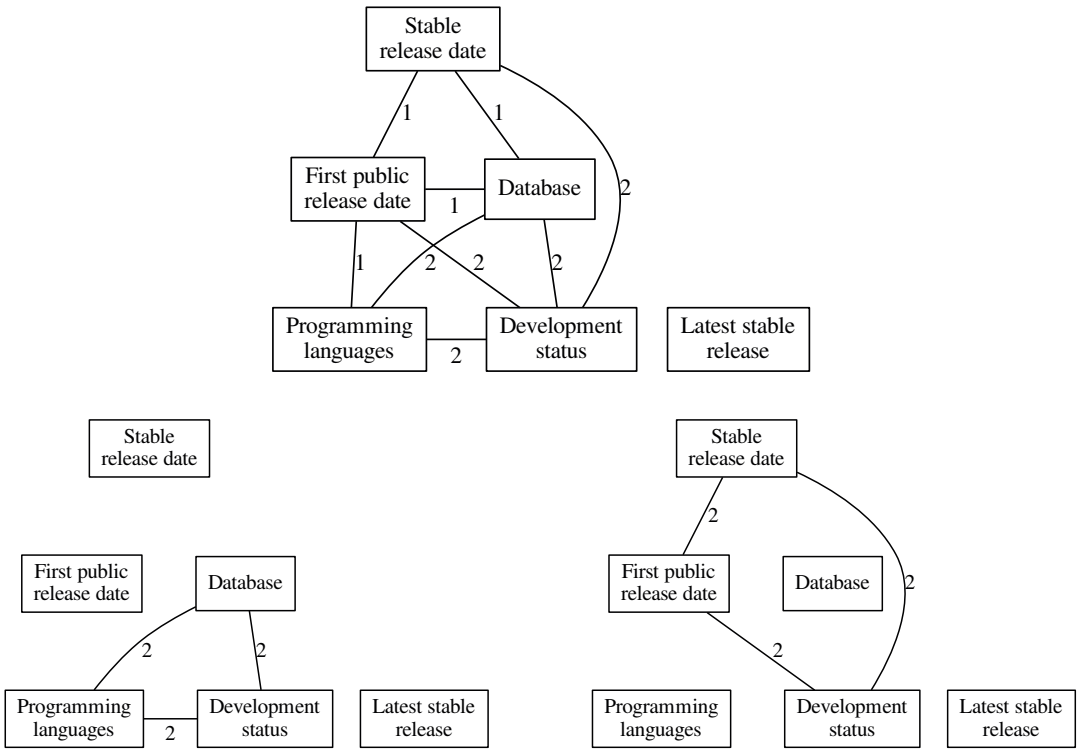


FIGURE 5.20 – Influence des caractéristiques des logiciels de comptabilité : générale (haut), préoccupation 1 sur le développement (gauche), préoccupation 2 sur l'évolution (droite)

On peut y lire que :
— Dans tous les cas, la caractéristique `Latest stable release` qui indique le numéro de la dernière version stable n'influence aucune caractéristique.
— Dans le cas général, le langage de programmation peut influencer le type de la base de données.
— Dans le cas de l'évolution, on ne s'intéresse pas à l'influence des langages de programmation et du type de bases de données.
— Dans le cas général, la première date de sortie du logiciel peut influencer son type de base de données.
— Dans le cas du développement, on ne s'intéresse pas à l'influence de la première date de sortie sur le type de base de données.

Nous avons ensuite sélectionné la matrice `CRM_systems_0`, représentant des systèmes logiciels de gestion de relations clients. Nous avons défini une préoccupation, concernant le développement des logiciels et se concentrant sur les langages d'implémentation, la base de données et le système d'exploitation du serveur. Les deux graphes sont présentés dans la Figure 5.21.

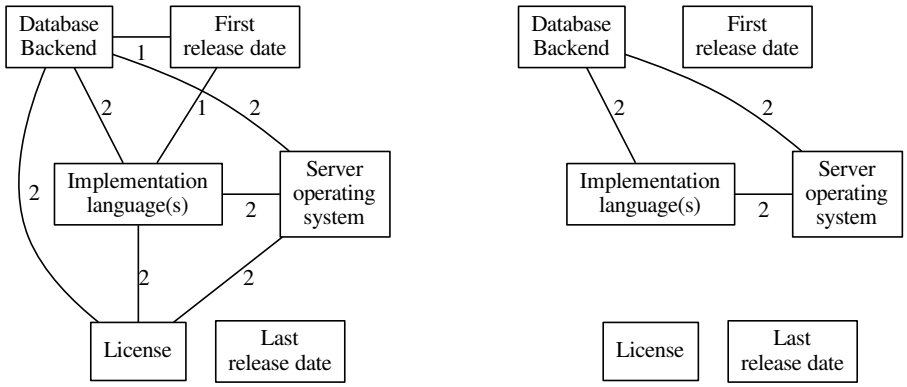


FIGURE 5.21 – Influence des caractéristiques des systèmes CRM : générale (gauche), préoccupation sur le développement (droite)

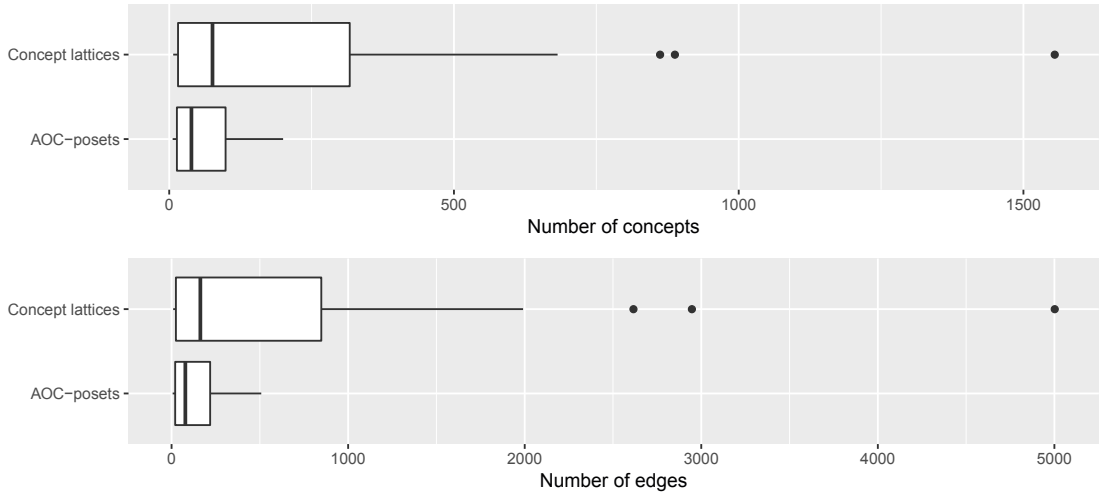
On peut y lire que :
— Dans tous les cas, on s'intéresse à l'influence entre le langage de programmation et le type de base de données.
— Dans le cas général, nous pensons que la date de sortie et le langage d'implémentation peuvent éventuellement s'influencer.
— Dans le cas du développement, on ne s'intéresse pas à l'influence du choix du système d'exploitation du serveur sur la licence du logiciel, contrairement au cas général.

RQ4. (employabilité de la méthode) Enfin, nous évaluons l'employabilité de la méthode, c'est-à-dire si le nombre de relations extraites après les deux types de réduction est facilement manipulable par un expert. Pour cela, nous commentons les chiffres obtenus lors de la réponse aux questions **RQ2.** et **RQ3.**.

5.6.3 Analyse des résultats

RQ1. (applicabilité de la méthode) : La distribution des tailles des treillis de concepts et des AOC-posets associés aux 27 MPCs étudiées est présentée à la Figure 5.22.

La construction des structures a été réalisée avec `RCAExplore`, sur un ordinateur *Intel(R) Core(TM) i7-6700HQ CPU@2.60GHz, 16GiB RAM*. `RCAExplore` indique le temps



	min.	q1	q2	mean	q3	max.
#Concepts (Lattices)	7	16	76	250	317	1555
#Concepts (AOC-posets)	6	14	39	58	99	200
#Edges (Lattices)	9	25	163	732	848	5049
#Edges (AOC-posets)	6	21	78	136	219	508

FIGURE 5.22 – Distribution de la taille des treillis de concepts et des AOC-posets pour les 27 MCPs étudiées

(en secondes) qu'il lui a été nécessaire pour construire la structure. Malgré le *scaling* binaire des taxonomies créant des contextes formels de taille importante, la construction des structures est quasi instantanée (entre 0 et 1 seconde). On observe la présence de données aberrantes dans la Figure 5.22 (points à l'extrémité des diagrammes). Elles représentent des cas isolés où les treillis obtenus sont beaucoup plus grands que la moyenne, jusqu'à 6 à 7 fois supérieurs dans notre étude. Même dans ces cas-là, la construction de la structure ne dépasse pas 1 seconde. Les fichiers `.dot` représentant les concepts et leur relation d'ordre qui sont produits par `RCAExplore`, peuvent être explorés manuellement. De plus, leur manipulation par le logiciel développé n'a pas été freinée par leur taille ni par leur complexité.

On observe dans un second temps que le gain de l'utilisation des AOC-posets comme alternative aux treillis de concepts est important. En moyenne, un AOC-poset représente 4 fois moins de concepts qu'un treillis et 5 fois moins d'arêtes. On voit aussi que, contrairement aux treillis, les AOC-posets n'ont pas de données aberrantes représentant des structures anormalement grandes par rapport à la tendance. Dans le pire des cas, les AOC-posets ont 8 fois moins de concepts et 10 fois moins d'arêtes. Cela s'explique par le fait que le nombre de concepts peut croître exponentiellement dans un treillis, ce qui n'est pas le cas pour les AOC-posets.

Conclusions : La taille des structures conceptuelles associées aux structures de patterns issues de MCPs ne limite pas leur construction ni leur manipulation. De plus, l'utilisation des AOC-posets prodigue une réduction importante des données manipulables, sans pour autant perdre de l'information concernant la variabilité. La méthode proposée est donc parfaitement applicable sur ce type de descriptions de produits.

RQ2. (gain de l'élimination de la redondance) : Les Tables 5.5 et 5.6 représentent les nombres de relations extraites pour chacune des 28 MCPs étudiées. La Table 5.5 présente

les cooccurrences (`Cooc`) et les mutex (`Mut`) et la Table 5.6 la fermeture transitive (`TC`) et la réduction transitive (`TR`) des implications. Pour chaque type de relations, nous avons d'abord conduit l'extraction sans utiliser de taxonomies, i.e., en appliquant un *scaling* simple (colonnes notées $\not T$), puis en utilisant des taxonomies (colonnes notées `T`) et enfin avec des taxonomies et les heuristiques de réduction de la redondance (colonnes notées `T+R`), excepté pour les cooccurrences, sur lesquelles nous n'avons pas défini d'heuristiques.

Ici, nous cherchons uniquement à commenter les différences entre les trois méthodes d'extraction en termes de nombre de relations extraites. Pour faciliter la visualisation des différences entre l'utilisation ou non de taxonomies, nous avons coloré les cellules des deux tables présentées : les cellules des colonnes représentant le *scaling* simple (sans taxonomies) apparaissent en bleu ; les cellules des autres types de colonnes correspondant à l'utilisation de taxonomies (`T` et `T+R`) apparaissent en bleu si le nombre de relations extraites est le même que celui obtenu sans utiliser de taxonomies, en vert si le nombre de relations est inférieur et en rouge s'il est supérieur. Par exemple, la ligne `accounting_software_1` de la Table 5.5 montre que :

- avec ou sans taxonomies, on extrait le même nombre de cooccurrences, i.e., 2 : les deux cellules sont donc bleues ;
- sans taxonomies, on extrait 63 mutex ;
- si l'on utilise des taxonomies, on extrait 80 mutex : ce nombre est supérieur à 63, la cellule est donc rouge ;
- si l'on applique les heuristiques d'élimination de la redondance sur les 80 mutex extraits avec des taxonomies, on obtient 30 mutex : ce nombre est inférieur aux 63 de départ, obtenus avec un *scaling* simple, la cellule est donc verte.

PCM's name	($\not T$) Cooc	(T) Cooc	($\not T$) Mut	(T) Mut	(T+R) Mut
ex_redac	4	5	29	30	19
accounting_software_0	13	13	455	455	241
accounting_software_1	2	2	63	80	30
accounting_software_2	34	35	1562	4253	405
ADC_software_0	0	0	12	19	4
ADC_software_1	6	6	74	88	7
ADC_software_2	0	0	26	26	5
ADC_software_3	5	5	38	49	13
ADC_software_4	0	0	3	3	0
ADC_software_5	0	0	0	0	0
ADC_software_6	1	1	34	41	17
ADC_software_7	25	25	145	151	23
ADC_software_8	1	1	22	22	1
ADC_software_9	0	0	10	15	5
ADC_software_10	0	0	34	36	19
ADC_software_11	1	2	11	16	6
AMD_chipsets_1	20	27	1503	1953	802
Android_e-book_reader_software_0	6	6	474	970	192
antivirus_software_0	21	23	4573	25513	1877
audio_player_software_0	7	7	496	1722	166
3D_computer_graphics_software_0	13	13	804	1736	143
antivirus_software_1	3	6	342	719	158
application_servers_0	12	29	909	2269	240
assemblers_5	3	4	165	207	65
business_integration_software_0	6	7	1238	1617	478
command_shells_0	40	41	1712	5574	760
CRM_systems_0	15	15	1644	6619	419
file_verification_software_0	1	1	405	2965	135
Mean	8.53	9.78	599	2041	222.5

TABLE 5.5 – Nombre de cooccurrences et de mutex extraits pour chacune des 28 MCPs étudiées, sans utiliser de taxonomies ($\not T$), en utilisant des taxonomies (T) et en utilisant des taxonomies et les heuristiques d'élimination de la redondance (T+R)

- *Cooccurrences* : Les colonnes 2 et 3 de la Table 5.5 montrent le nombre de cooccur-

rences extraites. Comme indiqué précédemment, aucune heuristique d'élimination de la redondance n'est définie sur ce type de relations. On peut voir que dans la majeure partie des cas, l'utilisation de taxonomies n'augmente pas le nombre de cooccurrences extraites. Dans les cas où l'on note une augmentation, elle est bien souvent dérisoire (5 au lieu de 4 pour `accounting_software_0`, 41 au lieu de 40 pour `command_shells_0`). Très rarement, on note une augmentation un peu plus forte (29 au lieu de 12 pour `application_servers_0`). En moyenne, l'utilisation de taxonomies extrait 9.7 cooccurrences, contre 8.5 sans taxonomies. Cette légère augmentation est due à l'introduction de nouvelles abstractions qui peuvent apparaître en même que d'autres éléments dans chaque description de produits.

◦ *Mutex* : Les colonnes 4, 5 et 6 de la Table 5.5 présentent le nombre de mutex extraits. L'introduction de taxonomies dans la méthode d'extraction augmente dans la grande majorité des cas le nombre de mutex extraits. Cette augmentation peut être faible (36 au lieu de 34 pour `ADC_software_10`), ou bien considérable (25513 au lieu de 4573 pour `antivirus_software_0`). En moyenne, 241% de mutex sont extraits en plus. L'élimination de la redondance donne de très bons résultats : dans tous les cas où l'on extrait des mutex, leur nombre est moins important lorsque l'on utilise des taxonomies et les heuristiques de réduction. En moyenne l'élimination de la redondance permet d'écarter 89% des mutex extraits en utilisant des taxonomies et on garde un nombre de mutex équivalent à 37% des mutex extraits sans taxonomies.

PCM's name	(∄) TC	(T) TC	(T+R) TC	(∄) TR	(T) TR	(T+R) TR
ex_redac	26	51	17	13	26	10
accounting_software_0	521	521	484	138	138	137
accounting_software_1	38	41	24	29	32	23
accounting_software_2	245	2247	187	155	352	150
ADC_software_0	7	7	7	7	7	7
ADC_software_1	40	42	26	14	15	11
ADC_software_2	7	7	4	7	7	4
ADC_software_3	72	74	28	20	22	15
ADC_software_4	6	6	6	4	4	4
ADC_software_5	5	5	5	5	5	5
ADC_software_6	20	22	17	15	17	15
ADC_software_7	313	325	85	72	78	42
ADC_software_8	20	20	20	12	12	12
ADC_software_9	11	12	12	6	7	7
ADC_software_10	7	11	10	7	9	9
ADC_software_11	11	13	10	9	10	8
AMD_chipsets_1	316	816	260	126	175	138
Android_e-book_reader_software_0	165	824	163	84	147	84
antivirus_software_0	948	20840	887	389	920	402
audio_player_software_0	84	1484	86	65	195	75
3D_computer_graphics_software_0	194	998	85	136	216	89
antivirus_software_1	254	901	230	83	138	77
application_servers_0	104	1483	147	75	199	82
assemblers_5	41	43	27	37	39	27
business_integration_software_0	126	1617	135	84	198	106
command_shells_0	496	4117	410	200	397	195
CRM_systems_0	232	5027	185	155	445	169
file_verification_software_0	84	2701	78	55	242	56
Mean	156.89	1580	129.82	71.5	144.71	69.96

TABLE 5.6 – Nombre d'implications extraites dans la fermeture transitive (TC) et la réduction transitive (TR) pour chacune des 28 MCPs étudiées, sans utiliser de taxonomies (∄), en utilisant des taxonomies (T) et en utilisant des taxonomies et les heuristiques d'élimination de la redondance (T+R)

◦ *Fermeture transitive* : Les colonnes 2, 3 et 4 de la Table 5.6 montrent la fermeture transitive des implications binaires extraites des 28 MCPs. On peut voir que l'introduction de nouvelles abstractions à travers les taxonomies fait croître fortement le nombre de relations extraites en moyenne : 907% de relations sont extraites en plus. Là encore, l'augmentation

est parfois assez faible (41 au lieu de 38 pour `accounting_software_1`), mais peut atteindre des nombres conséquents (20840 au lieu de 948 pour `antivirus_software_0`, 5027 au lieu de 232 pour `CRM_systems_0`). Cette augmentation est entre autres due à l'introduction de nouvelles abstractions, pouvant potentiellement produire de nouvelles implications De plus, les taxonomies représentent des implications entre leurs éléments (depuis les valeurs les plus spécifiques vers les valeurs les plus générales) qui se retrouvent dans le treillis de patterns et qui sont donc comptabilisées dans les implications extraites. L'élimination de la redondance permet d'écarter 82% des implications extraites avec les taxonomies et permet d'obtenir un ensemble d'implications dont la taille représente 83% du nombre d'implications extraites sans taxonomies. On note que dans 5 cas, malgré l'utilisation des heuristiques de réduction, on obtient un nombre d'implications binaires plus important qu'avec le *scaling* simple : cela est dû au fait que les taxonomies introduisent des nouvelles abstractions et font apparaître des implications qui ne peuvent pas être extraites avec un *scaling* simple. Cependant, même si le nombre d'implications binaires est plus important dans ces cas-là, il ne l'est pas de beaucoup : le pire des cas est pour la MCP `application_servers_0`, pour laquelle on obtient 147 implications après élimination de la redondance, contre 104 sans utilisation de taxonomies. En contre-partie de cette augmentation, on extrait de nouvelles implications binaires comprenant des valeurs plus générales sur les attributs.

◦ *Réduction transitive* : Les colonnes 5, 6 et 7 de la Table 5.6 présentent la réduction transitive des implications binaires pouvant être extraites des 28 MCPs. L'utilisation des taxonomies fait naturellement croître le nombre de relations extraites par rapport à un *scaling* simple, mais contrairement à la fermeture transitive, cette augmentation est moins considérable : 101% de relations sont extraites en plus, contre 907% avec la fermeture transitive. Dans les deux tiers des cas, l'application des heuristiques d'élimination de la redondance permet d'obtenir un nombre d'implications binaires plus petit que celui obtenu sans utilisation de taxonomies. En moyenne, l'élimination de la redondance écarte 52% des implications binaires extraites avec les taxonomies, mais le nombre de relations d'implications binaires ainsi obtenu est proche de celui obtenu sans l'utilisation de taxonomies. Les cas où le nombre obtenu après l'application des heuristiques est plus grand que celui obtenu sans taxonomies, la différence n'est pas très importante (le pire des cas étant 402 contre 389 pour `antivirus_software_0`).

La moyenne des relations extraites avant et après l'élimination de la redondance et avec un *scaling* binaire simple est illustrée dans la Figure 5.23.

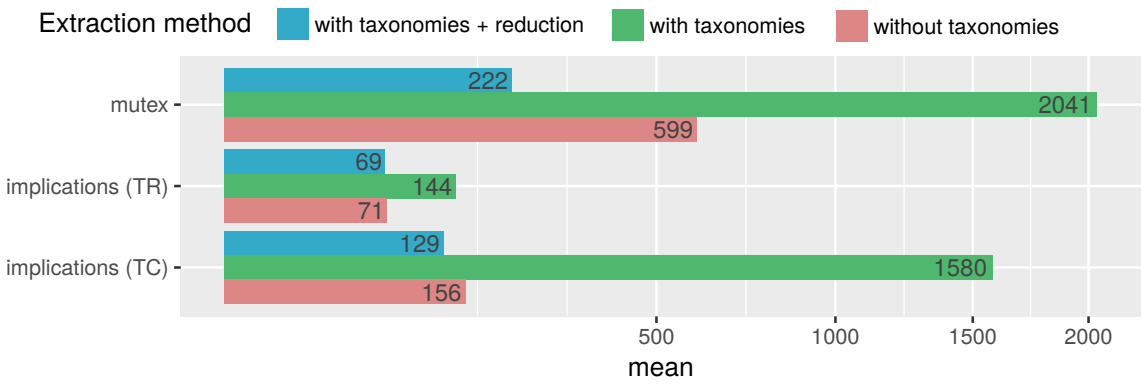


FIGURE 5.23 – Moyennes des relations extraites, en utilisant des taxonomies avant et après l'élimination de la redondance et sans utiliser de taxonomies

Conclusions : Comme nous le prédisions, l'utilisation de taxonomies fait augmenter

TABLE 5.7 – Distribution des relations extraites des 27 MCPs

	min.	q1	q2	mean	q3	max.
(⊤) Mutex	0	28.25	155.0	599.4	830.2	4573
(T) Mutex	0	29.0	179.0	2041.0	1790.0	25510
(T) Reduced mutex	0	6.7	47.5	222.5	240.2	1877
(⊤) Implications (TC)	5	17.8	78.0	156.9	235.2	948
(T) Implications (TC)	5	18.25	199.5	1581.0	1483.0	20840.0
(T) Reduced implications (TC)	4	15.75	53.0	129.8	168.5	887
(⊤) Implications (TR)	4	11.25	46.0	71.5	94.5	389
(T) Implications (TR)	4	11.5	58.5	144.7	198.2	920
(T) Reduced implications (TR)	4	9.75	34.5	69.9	93.2	402
(⊤) Cooccurrences	0	1.0	4.5	8.5	13.0	40
(T) Cooccurrences	0	1.0	5.5	9.7	13.5	41

le nombre de relations extraites du fait de l'introduction de nouvelles valeurs, représentant des abstractions des valeurs d'attributs. Cette augmentation est très importante dans le cas des mutex et de la fermeture transitive des implications binaires. Elle reste faible pour les cooccurrences et la réduction transitive des implications binaires. Dans le cas des mutex et des implications binaires, l'utilisation des taxonomies puis l'application des heuristiques de réduction de la redondance permettent d'obtenir en moyenne un nombre de relations moins important que si aucune taxonomie n'est utilisée. Dans certains cas, on extrait plus d'implications binaires (fermeture et réduction transitive) malgré l'élimination de la redondance. Cependant, cette augmentation reste assez faible et en contre-partie, de nouvelles implications ne pouvant pas être mises en évidence sans l'utilisation de taxonomies sont extraites. Cette méthode permet donc d'extraire des informations plus fines sur la variabilité, tout en les présentant de manière plus compacte.

RQ3. (pertinence des relations extraites) : La Figure 5.24 présente la répartition de la pertinence des relations extraites de la MCP `accounting_software_2` en fonction des trois graphes d'influence de la Figure 5.20. On peut voir que dans le cas général, 70% des mutex, 62% des cooccurrences et 43% des implications sont notées comme étant non pertinentes. En tout, c'est 60% de l'ensemble des relations extraites qui sont notées comme non pertinentes.

Ce nombre est encore plus important lorsque l'on considère des préoccupations spécifiques : 89 % des relations sont non pertinentes lorsque l'on s'intéresse à l'évolution et 85% pour la préoccupation de développement.

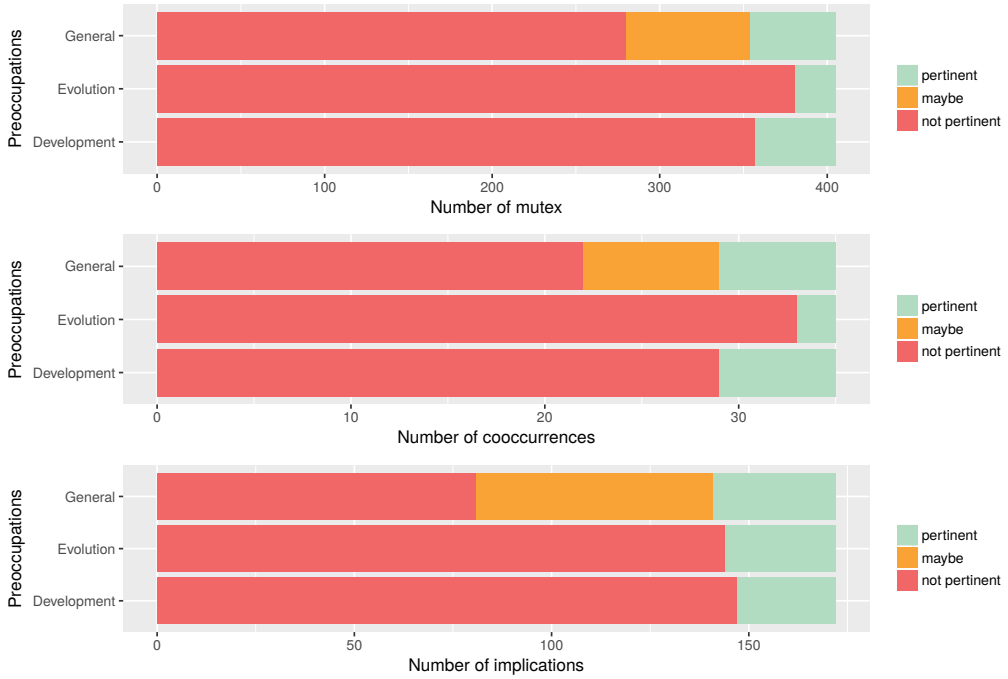


FIGURE 5.24 – Après filtrage de pertinence : mutex, cooccurrences et implications (sans réduction transitive) de la MCP `accounting_software_2`

La Figure 5.25 présente cette fois-ci la répartition de la pertinence des relations extraites depuis la MCP `CRM_systems_0` en fonction des deux graphes de la Figure 5.21. Dans le cas général, 35% des relations sont notées non pertinentes et environ 70% lorsque l'on s'intéresse au développement.

Conclusions : Une grande majorité des relations extraites ne sont pas pertinentes du point de vue du domaine. Le filtrage des relations par graphe d'influence permet d'en réduire considérablement le nombre. Cela montre la nécessité de développer des méthodes de "filtrage" plus poussées en complément des méthodes d'extraction et de filtrage présentées ici.

RQ4. (employabilité de la méthode) : On extrait en moyenne 10 cooccurrences avec des taxonomies : ce type de relation peut donc facilement être inspecté par un expert du fait de leur nombre réduit. Même dans le pire des cas, seulement 41 cooccurrences sont extraites. Le nombre de cooccurrences extraites avec et sans taxonomies ne change que très légèrement : seulement deux de plus en moyenne si aucune taxonomie n'est utilisée, du fait de la stratégie de *scaling* binaire des MCPs. Les graphes d'influence n'éliminent que peu de cooccurrences, mais l'élimination des relations accidentelles est ici moins cruciale du fait du nombre initialement réduit de cooccurrences.

Les implications binaires extraites (fermeture transitive) sont plus nombreuses : 130 en moyenne par MCP lorsque l'on utilise des taxonomies et les heuristiques d'élimination de la redondance et 157 lorsque l'on n'en utilise pas. La médiane est seulement de 53 après élimination de la redondance : la moitié des MCPs étudiées ont donc moins de 50 implications extraites pour la fermeture transitive. On peut voir qu'il y a des cas où l'extraction est vraiment trop importante : jusqu'à 887 implications extraites dans le pire des cas. On note de plus qu'un quart des MCPs étudiées ont plus de 1483 implications extraites, mais plus que 169 après élimination de la redondance. On en garde donc en

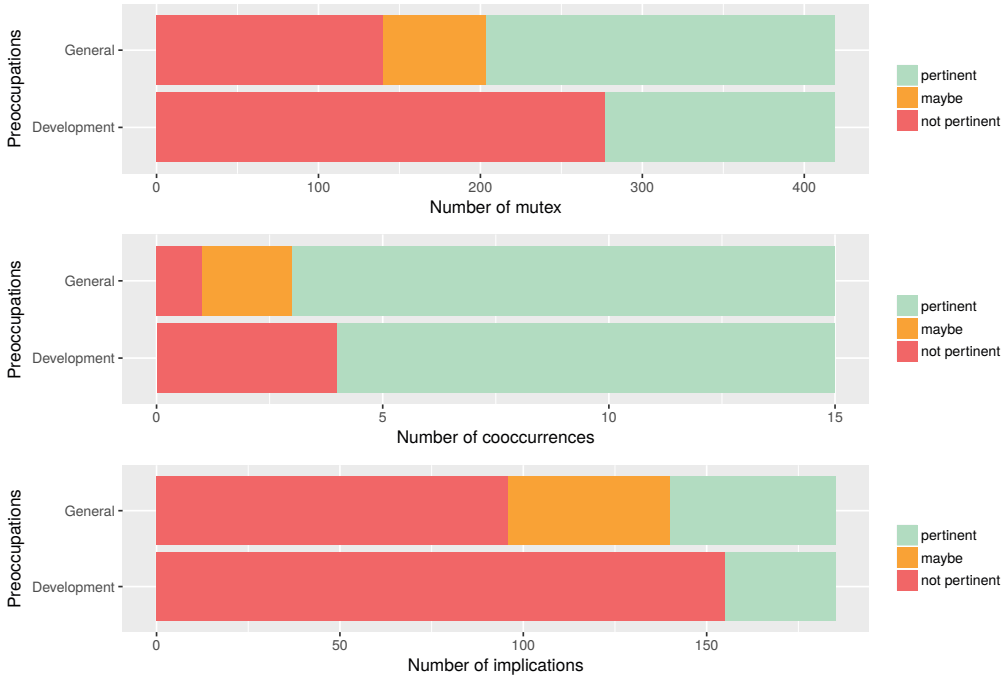


FIGURE 5.25 – Après filtrage de pertinence : mutex, cooccurrences et implications (sans réduction transitive) de la MCP `CRM_systems_0`

moyenne 25% sans perte d'information, mais il en reste en moyenne un nombre assez important pour qu'elles soient difficiles à étudier manuellement. Cependant, il semblerait que le pourcentage d'implications binaires représentant des relations accidentelles avoisine les 50%, ce qui pourrait rendre l'étude manuelle de certains cas facilement réalisable.

La réduction transitive de l'ensemble des implications extraites diminue par 10 leur nombre en moyenne. On en extrait 144 en moyenne par MCP, mais plus que 69 après élimination de la redondance : leur étude manuelle est donc faisable.

Malgré la réduction considérable du nombre de mutex grâce à l'élimination de la redondance, il en reste en moyenne plus de 222 à étudier. Leur nombre important est dû au fait que certains attributs ont des valeurs qui varient beaucoup, i.e., pour un attribut donné, chaque produit a une valeur unique ou bien qui n'est partagée que par peu d'autres produits. Ainsi, un nombre important de valeurs d'attributs différents n'apparaissent jamais ensemble dans un ensemble de descriptions de produits, ce qui se traduit par de nombreuses exclusions mutuelles ; celles-ci ne sont pas très intéressantes à étudier car elles sont majoritairement accidentelles. Cela se confirme lorsque l'on étudie les Figures 5.24 et 5.25.

Dans le pire des cas, même si la réduction est importante, beaucoup de relations restent à étudier : jusqu'à 1877 pour les mutex, 887 pour la fermeture transitive et 402 pour la réduction transitive. Sans filtrage additionnel, les modèles de variabilité synthétisés depuis ces relations possèderaient un nombre trop important de contraintes transverses comparé à la taille de la hiérarchie de caractéristiques, diminuant ainsi leur lisibilité et leur compréhensibilité.

Conclusions : Cette méthode extrait un nombre important de relations, rendant difficile leur étude manuelle (exceptées les cooccurrences). L'élimination de la redondance dans les relations extraites permet d'en réduire fortement le nombre, mais elles peuvent

rester nombreuses dans certains cas, notamment les mutex et la fermeture transitive des implications. Le filtrage par pertinence diminue encore ce nombre, mais des travaux plus poussés sur ce sujet restent à réaliser.

Des techniques de filtrage automatiques ou semi-automatiques complémentaires sont nécessaires pour aider les experts à étudier les relations obtenues.

5.6.4 Obstacles potentiels à la validité de l'évaluation

Validité externe Un obstacle potentiel à la validité externe est que notre évaluation qualitative et quantitative est basée sur une trentaine de MCPs, ce qui n'est possiblement pas assez représentatif des descriptions de produits en général. Le fait de devoir nettoyer manuellement chaque MCP a limité le nombre de descriptions étudiées. Cependant, des travaux récents s'intéressent à l'extraction et l'analyse automatique de MCPs [NBA+17]. Ce genre de travaux pourraient faciliter le processus d'obtention de MCPs au bon format et permettre de conduire des évaluations sur un nombre plus important de MCPs. Il est à noter que les MCPs de Wikipedia peuvent ne pas être assez représentatives des MCPs provenant de sources différentes.

Validité de construction Un obstacle potentiel à la validité de la construction de l'expérimentation est lié à l'heuristique utilisée pour identifier les relations "accidentelles" parmi celles qui sont extraites. Il est en effet possible que certaines des relations notées comme pertinentes ne soient pas utiles pour modéliser la variabilité de la FPL, ou bien que certaines soient plus importantes que d'autres. Cela montre la nécessité d'approfondir l'analyse des relations extraites en complément de notre approche, pour tendre vers le développement d'un outil efficace pour aider les experts à produire des modèles de variabilité depuis des descriptions de produits.

Un dernier obstacle potentiel de construction est la validité des algorithmes développés et utilisés pour réaliser l'implémentation. Les algorithmes d'extraction s'appuient sur des propriétés formelles connues des structures de l'AFC : le véritable obstacle potentiel réside donc dans leur implémentation. Pour limiter cet obstacle, nous avons étudié manuellement les résultats de chaque étape, d'abord sur de petits exemples, puis sur certaines des MCPs sélectionnées.

En résumé (5.6) :

- La **taille des structures conceptuelles** associées aux structures de patterns issues de MCPs, qui sont au cœur de la méthode proposée **ne limite pas leur construction et leur manipulation** : la méthode est donc applicable.
- Cette méthode extrait un nombre important de relations, rendant difficile leur étude manuelle, surtout pour les **mutex** et les **implications**. Mais les approches de **réduction de la redondance** grâce aux taxonomies réduisent considérablement le nombre de relations à étudier.
  - Les **mutex** restent très nombreux, malgré la réduction de la redondance.
  - La **fermeture transitive des implications** est trop considérable pour être étudiée manuellement. Mais sa **réduction transitive est employable**.
  - Les **cooccurrences** sont peu nombreuses en moyenne et elles sont donc facilement exploitables.
- Une grande majorité des relations extraites sont identifiées comme étant **accidentelles**, montrant la nécessité de mettre en place des solutions complémentaires pour aider à filtrer ces relations.

5.7 Conclusion

Dans ce chapitre, nous avons mis en place un processus pour extraire des relations logiques caractérisant la variabilité étendue à partir de matrices de comparaison de produits.

Nous avons d'abord étudié trois extensions des FMs et nous avons défini leur sémantique logique pour caractériser les relations logiques correspondant à la variabilité étendue. Nous avons ensuite défini les structures de patterns, une généralisation de l'analyse formelle de concepts permettant de traiter des données plus complexes que des attributs booléens. Nous avons proposé d'utiliser des vecteurs de patterns hétérogènes comme données d'entrée pour représenter des descriptions de produits complexes. Les structures de patterns permettent d'extraire toutes les relations logiques (parmi celles caractérisées précédemment) qui sont vraies pour les données considérées.

Le processus d'extraction a été appliqué aux matrices de comparaison de produits, des descriptions de produits complexes mais peu formalisées que l'on trouve en abondance sur internet. Nous avons défini un format pour les matrices de comparaison de produits afin qu'elles puissent être traitées automatiquement. Cette étape de prétraitement et de nettoyage a été réalisée manuellement. Nous avons ensuite proposé une méthode pour représenter une matrice de comparaison sous la forme de structures de patterns, en composant les valeurs des caractéristiques de chaque produit dans un vecteur de patterns hétérogènes. Des pistes pour réaliser automatiquement cette transformation, grâce à la construction automatique de taxonomies à partir des valeurs de la MCP, ont été données. L'application de l'analyse formelle de concepts sur les structures de patterns ainsi obtenues a permis d'extraire les relations logiques recherchées de manière automatique.

Nous avons réalisé une évaluation quantitative et qualitative de la méthode proposée sur environ 30 matrices de comparaison pour estimer son applicabilité, son employabilité et sa pertinence. L'applicabilité a été évaluée en construisant les structures conceptuelles au cœur de notre approche et en en analysant leur taille. L'employabilité a été estimée en étudiant le nombre de relations extraites qui pourraient être manipulées par un expert. La pertinence a été évaluée en détectant le pourcentage de relations accidentelles (vraies pour les descriptions considérées mais fausses du point de vue du domaine) extraites par notre approche grâce à une heuristique. Nos expérimentations ont montré que notre méthode passe à l'échelle même lorsqu'elle est appliquée sur des données de taille importante issues de wikipedia. Elle extrait un grand nombre de relations logiques, mais les structures de patterns et leurs taxonomies permettent d'en éliminer la redondance et de réduire rigoureusement le nombre. Enfin, elle nécessite d'être complétée avec d'autres approches permettant de filtrer les relations, de les ordonner et d'en évaluer plus minutieusement la pertinence.

Chapitre 6

Conclusion et Perspectives

Préambule

Dans ce chapitre, nous présentons d'abord une synthèse des travaux réalisés dans cette thèse qui montrent que l'analyse formelle de concepts est un cadre structurel de représentation de la variabilité (Chapitre 3) qui est réutilisable (Chapitre 4) et extensible (Chapitre 5). Nous discutons ensuite de quatre perspectives de ces travaux. La première concerne l'étude de la variabilité complexe dans le contexte de l'ingénierie des lignes de produits logiciels complexes grâce aux extensions de l'analyse formelle de concepts. La deuxième aborde la nécessité d'établir un parallèle avec l'ingénierie dirigée par les modèles et la modélisation des familles de produits logiciels sur plusieurs niveaux d'abstraction. La troisième s'intéresse à la représentation des connaissances et à l'association entre l'analyse formelle de concepts et les ontologies pour compléter les travaux actuels. Enfin, la quatrième concerne l'approfondissement des expérimentations réalisées et le développement d'un outil implémentant les méthodes proposées.

Sommaire

6.1 Synthèse des travaux

La thèse défendue dans ce manuscrit s'inscrit dans la nécessité de faciliter l'exploitation d'un ensemble de systèmes logiciels similaires, aussi appelé famille de produits logiciels.

Nous avons dans un premier temps étudié l'ingénierie des lignes de produits logiciels, un ensemble de méthodologies basées sur la réutilisation systématique et la personnalisation de masse, qui a pour but d'organiser l'effort de réutilisation et de faciliter l'exploitation des familles de produits logiciels. C'est un paradigme de développement qui s'articule autour d'une ligne de produits logiciels, une structure composée d'une architecture générique pouvant se combiner à un ensemble d'artefacts réutilisables, qui régente la variabilité des systèmes logiciels qui peuvent en être dérivés. L'adoption extractive de ces méthodologies est une pratique répandue qui consiste à construire une ligne de produits à partir d'un ensemble de variantes logicielles existantes dans le but d'en faciliter l'exploitation. La rétro-ingénierie de modèles de représentation de la variabilité a une importance cruciale dans le cadre de cette migration, car ils servent de support à la majorité des opérations assurant le fonctionnement de la ligne de produits. Cependant, la synthèse de ces modèles fait face à des défis pouvant gêner leur obtention ou bien le déroulement des opérations d'analyse et de gestion qu'ils encadrent.

Nous avons alors étudié les apports de l'analyse formelle de concepts durant la migration vers des lignes de produits logiciels, à travers son caractère structurel qui permet d'encadrer une partie de cette transition sans l'aide des modèles de variabilité standards.

Définie comme un cadre structurel représentant des caractéristiques fondamentales de la recherche d'information, l'analyse formelle de concepts produit des structures conceptuelles qui ont été reconnues comme des espaces d'information structurant la variabilité. Notre première contribution fut d'analyser l'aptitude des structures de base de l'AFC à représenter des modèles de variabilité. Pour cela, nous avons dans un premier temps approfondi les études existantes sur l'identification des informations de variabilité inhérentes aux quatre structures principales de l'AFC. Nous avons ensuite regroupé et unifié les méthodes d'extraction de la variabilité sous la forme de relations logiques, qui correspondent à la sémantique logique des modèles de caractéristiques définis dans FODA. Nous avons alors étudié les points communs entre les structures de base de l'AFC et les modèles de caractéristiques et montré qu'une structure conceptuelle représente une classe d'équivalence de tous les modèles de caractéristiques décrivant le même ensemble de configurations valides. Nous avons ainsi montré que les structures conceptuelles étaient des représentations naturelles de la variabilité, qui avaient des liens très forts avec les modèles de variabilité standards des LPLs.

Outre les informations classiques de la variabilité, nous avons identifié dans les structures conceptuelles des propriétés qui peuvent s'avérer utiles pour la gestion d'une FPL. Cela inclut notamment des connaissances sur les configurations, qu'elles soient partielles ou valides, et les nœuds de décisions relatifs à la construction de ces configurations. L'originalité de ce type d'informations est due au fait que les structures conceptuelles organisent à la fois les caractéristiques et les configurations, ce qui en fait un outil puissant pour travailler sur un ensemble de variantes existantes en ayant connaissance de leur variabilité intrinsèque. En nous appuyant là-dessus, nous avons montré l'aptitude des structures de l'AFC à être réutilisées comme support pour différentes opérations de gestion de la variabilité pouvant faciliter le processus de migration. Pour cela, nous avons proposé une implémentation originale de trois opérations connues, en nous basant sur l'AFC, qui sont la synthèse de modèles de caractéristiques, la composition de FPLs et la configuration de produits. Nous avons évalué chacune d'entre elles à travers une série d'expérimentations

montrant leur applicabilité ; ces expérimentations délimitent en partie les capacités de passage à l'échelle des structures conceptuelles et des algorithmes de construction associés. Nous avons montré que, même si ces structures n'ont pas toutes les capacités prêtées aux modèles de variabilité standards (incluant l'aspect ontologique et l'intelligibilité de la représentation), elles permettent tout de même de représenter, gérer et manipuler la variabilité de manière logique, structurelle et canonique. Elles offrent donc la possibilité d'appliquer certains traitements sur des FPLs n'ayant pas de modèles de caractéristiques standards.

Les informations sur la variabilité inhérentes à ces structures et les opérations qu'elles permettent de mettre en place peuvent évoluer et être étendues en même temps que le cadre structurel. L'AFC, initialement définie pour traiter des ensembles d'attributs binaires, a été étendue plusieurs fois afin de traiter des types de données bien plus complexes. Nous avons montré comment l'une de ces extensions, appelée structures de patterns, pouvait être appliquée dans le cadre de la migration pour prendre en compte des descriptions de variantes non plus booléennes mais multivaluées. Cette extension nous a permis de représenter et d'extraire des informations de variabilité étendue, correspondant à des modèles de variabilité plus complexes que les modèles de caractéristiques booléens traditionnels. C'est une première étape dans le processus de synthèse de modèles de caractéristiques étendus, dans le contexte de la migration vers des lignes de produits complexes. Parce que ce cadre peut s'adapter à plusieurs types de données, il permet à la représentation et à la gestion de la variabilité de s'adapter par la même occasion à d'autres types de problématiques.

L'analyse formelle de concepts est donc un cadre structurel qui met naturellement en évidence les informations fondamentales sur la variabilité des descriptions d'un ensemble de variantes ; les structures de base de l'AFC représentent un ensemble d'informations correct et complet vis-à-vis de la FPL étudiée. Elles sont un support réutilisable, pouvant encadrer un ensemble varié de traitements utiles lors de la migration. L'extension du cadre en lui-même n'invalide pas les informations intrinsèques sur la variabilité qu'il met en évidence, ni les traitements qu'il permet de mettre en œuvre ; il ouvre la voie de la rétro-ingénierie des lignes de produits logiciels complexes.

6.2 Perspectives

6.2.1 Lignes de produits complexes

Extraction de descriptions – Les travaux présentés ici se basent sur des descriptions plus ou moins complexes de variantes logicielles et présument l'existence de telles descriptions sans s'attarder sur les moyens de les obtenir. Afin de compléter ces travaux, une première perspective est de **mettre en place des méthodes pour extraire des descriptions (qu'elles soient simples ou élaborées) depuis du code source et d'autres artefacts**. Quelques approches basées sur l'AFC pour l'extraction et la localisation de caractéristiques dans le code source existent dans la littérature [XXJ12, SSD14]. Nous pensons qu'il serait intéressant de reprendre ces travaux et de les affiner en considérant les extensions de l'AFC permettant de prendre en compte des données d'entrée plus complexes. Ainsi, il serait possible d'analyser des systèmes eux aussi plus complexes, qui utilisent par exemple des technologies et des sous-systèmes hétérogènes et distribués de tailles différentes. L'analyse relationnelle de concepts [RHHNV13] et les structures de patterns [GK01] sont deux extensions qui nous semblent pertinentes pour innover dans la rétro-ingénierie de systèmes complexes. L'analyse relationnelle de concepts permet de définir des relations entre plusieurs ensembles de données et d'extraire des abstractions prenant en compte ces relations, ce qui motive l'utilisation de ce cadre pour l'extraction de descriptions caractérisant des systèmes-de-systèmes. Les structures de patterns permettent

de traiter des valeurs organisées dans des hiérarchies de généralisation/spécialisation : elles peuvent être utiles pour faire émerger de nouvelles abstractions et traiter des données de types hétérogènes.

Extraction de variabilité étendue – De la même façon que les extensions de l'AFC pourraient permettre d'élargir le périmètre de l'extraction des descriptions vers des données plus complexes, elles peuvent permettre de travailler sur des modèles de variabilité étendus extraits de ces descriptions. Une partie des travaux présentés ici portait sur l'utilisation des structures de patterns pour extraire des relations logiques représentant de la variabilité étendue, i.e., la représentation logique des modèles de variabilité étendus par des cardinalités et des attributs multivalués. Une autre perspective est de **développer ces travaux en y intégrant l'analyse relationnelle de concepts dans le but d'extraire des relations entre plusieurs modèles de variabilité, représentant des références entre des "familles" de sous-systèmes**. Une telle approche permettrait de mettre en place une séparation des préoccupations et de faciliter la gestion de FPLs de grande taille. La combinaison des deux extensions de l'AFC citées ci-dessus [CN14] pourrait donner lieu à des classes d'équivalence de modèles de variabilité étendus intégrant cardinalités, attributs et références [CHE05].

Niveaux d'abstraction – L'AFC et ses extensions peuvent encadrer l'extraction de descriptions complexes en mettant en évidence de nouvelles abstractions. Puis, ces descriptions pourront à leur tour être organisées dans des structures conceptuelles pour en analyser la variabilité. La migration depuis des artefacts logiciels vers des modèles de variabilité de haut niveau peut alors se résumer à **manipuler la granularité des structures conceptuelles pour raffiner des connaissances sur plusieurs niveaux d'abstraction**. Par exemple, nous pourrions structurer des morceaux de code source afin d'identifier des classes et des relations, qui seront à leur tour organisées dans le but de déterminer des fonctionnalités ou des caractéristiques de plus haut niveau. Enfin, structurer ces dernières permettrait de mettre en évidence leur variabilité et d'avoir des liens de traçabilité entre cette variabilité de haut niveau et le code source de départ. À long terme, la réalisation des deux objectifs précédents permettrait d'intégrer les méthodes d'extraction réalisées dans un seul et même cadre, permettant d'**accompagner un expert durant la migration depuis des codes sources de variantes logicielles vers des lignes de produits logiciels complexes**.

6.2.2 Ingénierie dirigée par les modèles

Métamodèles et niveaux d'abstraction – La place de l'ingénierie dirigée par les modèles dans la réalisation des objectifs précédents est très importante. Une perspective transversale serait d'**étudier des méta-modèles permettant de modéliser les systèmes logiciels complexes à plusieurs niveaux d'abstraction en parallèle des structures conceptuelles**. Il serait en effet intéressant d'étudier l'utilisation des modèles obtenus à partir de ressources telles que le code source et la documentation, incluant l'analyse des besoins, les décisions d'architecture ou encore la connaissance technique sur les styles et patterns employés, comme support aux méthodes de rétro-ingénierie mises en place pour obtenir des modèles de plus haut niveau, e.g., représentant les fonctionnalités du système de départ. Étudier des métamodèles existants tels que KDM (pour *Knowledge Discovery Metamodel*) ou FAMIX est fortement envisagé. Ces transformations de modèles-à-modèles, indépendantes du type de technologie et de l'architecture des variantes de départ, permettront aux étapes les plus abstraites de la migration d'être génériques et faciliteront l'extension du cadre global.

Alignement de métamodèles de variabilité – **Approfondir le parallèle entre les structures conceptuelles et différents modèles de variabilité** est une autre perspective de ces travaux, afin de faciliter le passage d'un formalisme à l'autre. Nous avons étudié les modèles de caractéristiques tels que définis dans FODA car ils sont les plus répandus, mais d'autres comparaisons avec CVL (*Common Variability Language*)[HWC13], les modèles de variabilité orthogonaux [PBvdL05], ou les modèles de caractéristiques étendus [CHE05] présentent leur importance pour rendre la migration plus générique. Ce genre de comparaisons pourrait de plus **renforcer nos travaux sur la pertinence des structures conceptuelles comme représentations de la variabilité**. Dans le même ordre d'idée, il serait intéressant d'explorer les autres informations sur la variabilité qui ne sont pas prises en compte par FODA mais qui sont mises en évidence dans les structures conceptuelles. Nous avons vu par exemple dans ce manuscrit un type d'informations que nous avons appelé *not all together*, représentant un ensemble de caractéristiques qui n'apparaissent jamais toutes ensemble en même temps. Nous souhaitons donc identifier d'autres types d'informations de variabilité et étudier leur pertinence pour la gestion d'une LPL.

6.2.3 Représentation des connaissances

Ontologies – Nous avons vu que l'aspect ontologique était crucial dans la modélisation de la variabilité, ce qui rendait souvent incomplètes les méthodes d'extraction automatique. Les structures conceptuelles pourraient, à l'instar des modèles de caractéristiques, grandement bénéficier d'une sémantique ontologique : une autre perspective serait donc de **combiner les structures conceptuelles avec des ontologies**. Les patterns structures permettant de manipuler des descriptions ayant la forme d'ontologies et l'analyse relationnelle de concepts définie sur la base d'une logique de description semblent être des outils pertinents pour améliorer cet aspect.

Systèmes de recommandation – À travers notre implémentation de la sélection de produits, nous avons étudié la capacité des structures conceptuelles à encadrer la navigation dans un espace d'informations structuré. Nous avons alors défini une méthode de génération locale d'une sous-hiérarchie des treillis de concepts, les AOC-posets, pour gérer cette opération. Depuis, nous avons étendu ces travaux pour réaliser de la génération locale de treillis de concepts connectés, obtenus par analyse relationnelle de concepts (voir Annexe). La navigation incrémentale dans des treillis connectés permettrait de gérer la séparation des préoccupations lors de la sélection de produits. Ainsi, chaque famille de sous-systèmes correspondrait à une structure, qui est organisée en fonction de ses propres caractéristiques et de ses relations avec les autres familles. Il serait alors possible de naviguer au sein d'une même structure ainsi que d'une structure à l'autre : nous souhaitons par la suite **étudier la navigation conceptuelle comme un moyen de sélectioner des produits dans les familles de produits complexes**. L'analyse formelle et relationnelle de concepts et les systèmes de recommandation ont déjà fait l'objet de travaux [Cod15] qu'il serait intéressant d'étudier et d'intégrer à nos perspectives.

6.2.4 Outillage et expérimentations

Implémentation – Une dernière perspective concerne l'**implémentation d'un démonstrateur logiciel regroupant les différentes méthodes et opérations proposées** dans la thèse et dans les perspectives. Un tel système permettrait d'approfondir les conclusions relatives aux expérimentations conduites dans les Chapitres 4 et 5. De plus, il serait possible d'aborder l'utilisation de tels outils du point de vue de l'utilisateur dans des expérimentations systématiques, ce qui compléterait nos observations.

Application – Pour finir, nous souhaiterions **appliquer les méthodes développées**

dans cette thèse à travers des collaborations avec des entreprises, afin d'en tester les limites et d'identifier de nouvelles questions de recherche. Un projet avec une architecte de systèmes d'informations d'IBM est en cours afin d'appliquer l'extraction de variabilité et l'utilisation de l'ECFD sur des problématiques concernant les lacs de données [ML16]. La génération locale dans les structures conceptuelles connectées est en cours d'implémentation dans `RCAExplore`, et est testée sur des données réelles sur les plantes à effet pesticide et antibiotique dans le cadre du projet `KnoMana` [1] [Ouz18].

1. http ://www.cirad.fr/en/news/all-news-items/articles/2017/science/identifying-plants-used-as-natural-pesticides-in-africa-knomana

Bibliographie

[ABH+13] Mathieu Acher, Benoit Baudry, Patrick Heymans, Anthony Cleve, and Jean-Luc Hainaut. Support for reverse engineering and maintaining feature models. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, pages 20 :1–20 :8, 2013.

[ABKS13] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

[ACLF10a] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Comparing approaches to implement feature model composition. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA'10)*, pages 3–19, 2010.

[ACLF10b] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Composing feature models. In *Revised Selected Papers of the 2nd International Conference on Software Language Engineering (SLE'09)*, volume 5969 of *LNCS*, pages 62–81. Springer, 2010.

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. Familiar : A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6) :657–681, 2013.

[ACP+12] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. On extracting feature models from product descriptions. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12)*, pages 45–54, 2012.

[ALN16] Mehwish Alam, Thi Nhu Nguyen Le, and Amedeo Napoli. LatViz : A New Practical Tool for Performing Interactive Exploration over Concept Lattices. In *Proceedings of the 13th International Conference on Concept Lattices and Their Applications (CLA'16)*, pages 9–20, 2016.

[AMdSH+14] Ra'Fat Al-Msie 'deen, Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Ahmad Al-Khlifat. Concept lattices : A representation space to structure software variability. In *Proceedings of the 5th International Conference on Information and Communication Systems (ICICS'14)*, pages 1–6, 2014.

[AMHS+14] Ra'Fat Al-Msie'deen, Marianne Huchard, Abdelhak Seriai, Christelle Urtado, and Sylvain Vauttier. Reverse Engineering Feature Models from Software Configurations using Formal Concept Analysis. In *Proceedings of the 11th International Conference on Concept Lattices and Their Applications (CLA'14)*, pages 95–106, 2014.

[AMSH+13] Ra'Fat Al-Msie'deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Mining features from the

object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering & Knowledge Engineering (SEKE'13)*, pages 244–249, 2013.

[BABN16] Guillaume Bécan, Mathieu Acher, Benoit Baudry, and Sana Ben Nasr. Breathing ontological knowledge into feature model synthesis : an empirical study. *Empirical Software Engineering*, 21(4) :1794–1841, 2016.

[Bat05] Don S. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, pages 7–20, 2005.

[BBGA15] Guillaume Bécan, Razieh Behjati, Arnaud Gotlieb, and Mathieu Acher. Synthesis of attributed feature models from product descriptions. In *Proceedings of the 19th International Conference on Software Product Line (SPLC'15)*, pages 1–10, 2015.

[BCK17] Alexandre Bazin, Jessie Carbonnel, and Giacomo Kahn. On-demand generation of aoc-posets : Reducing the complexity of conceptual navigation. In *Proceedings of the 23rd International Symposium on Foundations of Intelligent Systems (ISMIS'17)*, pages 611–621, 2017.

[BCNR17] Quentin Brabant, Miguel Couceiro, Amedeo Napoli, and Justine Reynaud. From Meaningful Orderings in the Web of Data to Multi-level Pattern Structures. In *Proceedings of 23rd International Symposium on Foundations (Methodologies) for Intelligent Systems (ISMIS'17)*, pages 622–631, 2017.

[BEJ+16] Aleksey Buzmakov, Elias Egho, Nicolas Jay, Sergei O. Kuznetsov, Amedeo Napoli, and Chedy Raïssi. On mining complex sequential data by means of FCA and pattern structures. *International Journal of General Systems*, 45(2) :135–159, 2016.

[Ber11] Karell Bertet. *Structure de treillis – Contributions structurelles et algorithmiques – Quelques usages pour des données images.* Habilitation à diriger des recherches (HDR), Université de La Rochelle, L3i, 2011.

[Beu12] Danilo Beuche. Modeling and building software product lines with pure : : variants. In *Proceedings of the 16th International Software Product Line Conference (SPLC'12)*, pages 255–255. ACM, 2012.

[BFGR13] David Benavides, Alexander Felfernig, José A. Galindo, and Florian Reinfrank. Automated analysis in feature modelling and product configuration. In *Proceedings of the 13th International Conference on Software Reuse (ICSR'13)*, pages 160–175, 2013.

[BGH+14] Anne Berry, Alain Gutierrez, Marianne Huchard, Amedeo Napoli, and Alain Sigayret. Hermes : a simple and efficient algorithm for building the aoc-poset of a binary relation. *Annals of Mathematics and Artificial Intelligence*, 72(1-2) :45–71, 2014.

[BGSS07] Franz Baader, Bernhard Ganter, Baris Sertkaya, and Ulrike Sattler. Completing description logic knowledge bases using formal concept analysis. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, volume 7, pages 230–235, 2007.

[BHM+05] Anne Berry, Marianne Huchard, Ross M. McConnell, Alain Sigayret, and Jeremy P. Spinrad. Efficiently Computing a Linear Extension of the Subhierarchy of a Concept Lattice. In *Proceedings of the 3rd International Conference on Formal Concept Analysis (ICFCA'05)*, pages 208–222, 2005.

[BM70] M Barbut and B Monjardet. L'ordre et la classification, algèbre et combinatoire, tome ii. paris, hachette, 1970. *Zbl0267*, 6001, 1970.

[BM11] Jorge Barreiros and Ana Moreira. Soft constraints in feature models. In *Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA'11)*, pages 136–141, 2011.

[BNT08] Rokia Bendaoud, Amedeo Napoli, and Yannick Toussaint. Formal concept analysis : A unified framework for building and refining ontologies. In *Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW'08)*, pages 156–171, 2008.

[Bos00] Jan Bosch. *Design and use of software architectures : adopting and evolving a product-line approach*. Pearson Education, 2000.

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. A survey of variability modeling in industrial practice. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, pages 7 :1–7 :8, 2013.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[BSA+14] Guillaume Bécan, Nicolas Sannier, Mathieu Acher, Olivier Barais, Arnaud Blouin, and Benoit Baudry. Automating the formalization of product comparison matrices. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering, (ASE'14)*, pages 433–444, 2014.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later : A literature review. *Information Systems*, 35(6) :615–636, 2010.

[BTN+08] Goetz Botterweck, Steffen Thiel, Daren Nestor, Saad bin Abid, and Ciarán Cawley. Visual tool support for configuring and understanding software product lines. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 77–86. IEEE, 2008.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the 17th International Conference of Advanced Information Systems Engineering (CAISE'05)*, pages 491–503, 2005.

[CBHN16] Jessie Carbonnel, Karell Bertet, Marianne Huchard, and Clémentine Nebut. FCA for software product lines representation : Mixing product and characteristic relationships in a unique canonical representation. In *Proceedings of the 13th International Conference on Concept Lattices and Their Applications (CLA'16)*, pages 109–122, 2016.

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative Programming for Embedded Software : An Industrial Experience Report. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, pages 156–172, 2002.

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool features and tough decisions : a comparison of variability modeling approaches. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12)*, pages 173–182, 2012.

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged Configuration Using Feature Models. In *Proceedings of the 3rd International Software Product Line Conference (SPLC'04)*, pages 266–283, 2004.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process : Improvement and Practice*, 10(1) :7–29, 2005.

[CHMN17] Jessie Carbonnel, Marianne Huchard, André Miralles, and Clémentine Nebut. Feature model composition assisted by formal concept analysis. In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'17)*, pages 27–37, 2017.

[CHN17] Jessie Carbonnel, Marianne Huchard, and Clémentine Nebut. Analyzing variability in product families through canonical feature diagrams. In *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering (SEKE'17)*, pages 185–190, 2017.

[CHN18] Jessie Carbonnel, Marianne Huchard, and Clémentine Nebut. On extracting relevant and complex variability information from software descriptions with pattern structures. In *Companion Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, pages 304–305, 2018.

[CKK06] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. In *Proceedings of the 10th International Software Product Line Conference (SPLC'06)*, pages 41–51, 2006.

[CN02] Paul Clements and Linda Northrop. *Software product lines : practices and patterns*, volume 3. Addison-Wesley Reading, 2002.

[CN14] Víctor Codocedo and Amedeo Napoli. A proposition for combining pattern structures and relational concept analysis. In *Proceedings of the 12th International Conference on Formal Concept Analysis (ICFCA'14)*, pages 96–111, 2014.

[CN15] Víctor Codocedo and Amedeo Napoli. Formal Concept Analysis and Information Retrieval -A Survey. In *Proceedings of the 13th International Conference on Formal Concept Analysis (ICFCA'15)*, pages 61–77, 2015.

[Cod15] Victor Codocedo-Henriquez. *Contributions to indexing and retrieval using Formal Concept Analysis. (Contributions à l'indexation et à la recherche d'information avec l'analyse formelle de concepts).* PhD thesis, University of Lorraine, Nancy, France, 2015.

[CR04] Claudio Carpineto and Giovanni Romano. Exploiting the Potential of Concept Lattices for Information Retrieval with CREDO. *Journal of Universal Computer Science*, 10(8) :985–1013, 2004.

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature Diagrams and Logics : There and Back Again. In *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*, pages 23–34, 2007.

[Cza99] Krzysztof Czarnecki. *Generative programming - principles and techniques of software engineering based on automated configuration and fragment-based component models.* PhD thesis, Technische Universität Illmenau, Germany, 1999.

[DDE08] Frithjof Dau, Jon Ducrou, and Peter W. Eklund. Concept Similarity and Related Categories in SearchSleuth. In *Proceedings of the 16th International Conference on Conceptual Structures (ICCS'08)*, pages 255–268, 2008.

[DDH+13] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. Feature model extraction from large collections of informal product descriptions. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 290–300, 2013.

[DDHL95] Hervé Dicky, Christophe Dony, Marianne Huchard, and Thérèse Libourel. ARES, Adding a class and REStructuring Inheritance Hierarchy. In *Informal proceedings of the Onzièmes Journées Bases de Données Avancées*, pages 25–42, 1995.

[DE07] Jon Ducrou and Peter W. Eklund. SearchSleuth : The Conceptual Neighbourhood of an Web Query. In *Proceedings of the 5th International Conference on Concept Lattices and Their Applications (CLA'07)*, 2007.

[DRB+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*, pages 25–34. IEEE, 2013.

[DVE06] Jon Ducrou, Björn Vormbrock, and Peter W. Eklund. FCA-Based Browsing and Searching of a Collection of Images. In *Proceedings of the 14th International Conference on Conceptual Structures (ICCS'06)*, pages 203–214, 2006.

[EGK+01] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, pages 483–484. Springer, 2001.

[Fer14] Sébastien Ferré. *Reconciling Expressivity and Usability in Information Access : from File Systems to the Semantic Web.* Habilitation à diriger des recherches (HDR), University of Rennes 1, 2014.

[FFA+13] Marianela Ciolfi Felice, João Bosco Ferreira Filho, Mathieu Acher, Arnaud Blouin, and Olivier Barais. Interactive visualisation of products in online configurators : a case study for variability modelling technologies. In *Workshop Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, pages 82–85, 2013.

[FR00] Sébastien Ferré and Olivier Ridoux. A logical generalization of formal concept analysis. In *Proceedings of the 8th International Conference on Conceptual Structures (ICCS'00)*, pages 371–384, 2000.

[FV03] David Faust and Chris Verhoef. Software product line migration and deployment. *Software : Practice and Experience*, 33(10) :933–955, 2003.

[Gan10] Bernhard Ganter. Two basic algorithms in concept analysis. In *Proceedings of the 8th International Conference on Formal Concept Analysis (ICFCA'10)*, pages 312–340, 2010.

[GF16] Gillian J. Greene and Bernd Fischer. Single-Focus Broadening Navigation in Concept Lattices. In *Proceedings of the 3rd Workshop on Concept Discovery in Unstructured Data, co-located with the 13th International Conference on Concept Lattices and Their Applications (CLA'16)*, pages 32–43, 2016.

[GGP89] Robert Godin, Jan Gecsei, and Claude Pichet. Design of a Browsing Interface for Information Retrieval. In *Proceedings of the 12th International Confe-*

rence on Research and Development in Information Retrieval (SIGIR'89), pages 32–39, 1989.

[GK01] Bernhard Ganter and Sergei O. Kuznetsov. Pattern Structures and Their Projections. In *Proceedings of the 9th International Conference on Conceptual Structures (ICCS'01)*, pages 129–142, 2001.

[GM93] Robert Godin and Hafedh Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 394–410, 1993.

[GO16] Bernhard Ganter and Sergei A. Obiedkov. *Conceptual Exploration*. Springer, 2016.

[Gru95] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing ? *International journal of human-computer studies*, 43(5-6) :907–928, 1995.

[GSG86] Robert Godin, Eugene Saunders, and Jan Gecsei. Lattice model of browsable data spaces. *Information Sciences*, 40(2) :89–116, 1986.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal concept analysis - mathematical foundations*. Springer, 1999.

[HGR12] Gerald Holl, Paul Grünbacher, and Rick Rabiser. A systematic review and an expert survey on capabilities supporting multi product lines. *Information & Software Technology*, 54(8) :828–852, 2012.

[HLE11] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. Reverse Engineering Feature Models from Programs' Feature Sets. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE'11)*, pages 308–312, 2011.

[HLE13] Evelyn Nicole Haslinger, Roberto Erick Lopez-Herrejon, and Alexander Egyed. On Extracting Feature Models from Sets of Valid Feature Combinations. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'13)*, pages 53–67, 2013.

[HP03] Günter Halmans and Klaus Pohl. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2(1) :15–36, 2003.

[Huc15] Marianne Huchard. Analyzing inheritance hierarchies through formal concept analysis : A 22-years walk in a landscape of conceptual structures. In *Proceedings of the 29th MechAnisms on SPEcialization, Generalization and inHerItance (MASPEGHI at ECOOP'15)*, pages 8–13, 2015.

[HWC13] Øystein Haugen, Andrzej Wąsowski, and Krzysztof Czarnecki. CVL : Common Variability Language. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, pages 277–277, 2013.

[JSvGB04] Anton Jansen, Rein Smedinga, Jilles van Gurp, and Jan Bosch. First class feature abstractions for product derivation. *IEE Proceedings - Software*, 151(4) :187–198, 2004.

[KAMN10] Mehdi Kaytoue-Uberall, Zainab Assaghir, Nizar Messai, and Amedeo Napoli. Two complementary classification methods for designing a concept lattice from interval data. In *Proceedings of the 6th International Symposium on Foundations of Information and Knowledge Systems (FoIKS'10)*, pages 345–362, 2010.

[KBB+01] Peter Knauber, Jesus Bermejo, Günter Böckle, Julio Cesar Sampaio do Prado Leite, Frank van der Linden, Linda Northrop, Michael Stark, and David M Weiss. Quantifying product line benefits. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering (PFE'01)*, pages 155–163. Springer, 2001.

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, 1990.

[KLD02] Kyo C Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented product line engineering. *IEEE software*, 19(4) :58–65, 2002.

[Kru01] Charles W. Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering (PFE'01)*, pages 282–293, 2001.

[Kru02] Charles W. Krueger. Practical strategies and techniques for adopting software product lines. In *Proceedings of the 7th International Conference on Software Reuse : Methods, Techniques, and Tools (ICSR'02)*, pages 349–350, 2002.

[Kru08] Charles W Krueger. The biglever software gears unified software product line engineering framework. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 353–353. IEEE, 2008.

[KTS+09] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide : A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 611–614. IEEE Computer Society, 2009.

[Kuz01] Sergei O. Kuznetsov. Machine learning on the basis of formal concept analysis. *Automation and Remote Control*, 62(10) :1543–1564, 2001.

[Leb00] Hervé Leblanc. *Sous-hiérarchies de Galois : un modèle pour la construction et l'évolution des hiérarchies d'objets*. PhD thesis, Université de Montpellier, 2000.

[LLE14] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Feature Model Synthesis with Genetic Programming. In *Proceedings of the 6th International Symposium on Search-Based Software Engineering (SSBSE'14)*, pages 153–167, 2014.

[LLG+15] Roberto Erick Lopez-Herrejon, Lukas Linsbauer, José A. Galindo, José Antonio Parejo, David Benavides, Sergio Segura, and Alexander Egyed. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, 103 :353–369, 2015.

[LP07] Felix Loesch and Erhard Ploedereder. Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 159–170, 2007.

[LS05] Lotfi Lakhal and Gerd Stumme. Efficient mining of association rules based on formal concept analysis. In *Formal concept analysis*, pages 180–195. Springer, 2005.

[Man02] Mike Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the 2nd International Software Product Line Conference (SPLC'02)*, pages 176–187, 2002.

[Mar06] Gary Marchionini. Exploratory search : from finding to understanding. *Communications of the ACM*, 49(4) :41–46, 2006.

[MBC09] Marcílio Mendonça, Moises Branco, and Donald D. Cowan. S.P.L.O.T. : software product lines online tools. In *Companion Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09)*, pages 761–762, 2009.

[MGA13] Cássio A. Melo, Bénédicte Le Grand, and Marie-Aude Aufaure. Browsing Large Concept Lattices through Tree Extraction and Reduction Methods. *International Journal of Intelligent Information Technologies (IJIIT)*, 9(4) :16–34, 2013.

[MHCN17] André Miralles, Marianne Huchard, Jessie Carbonnel, and Clémentine Nebut. Alignement, union et intersection de modèles : 3 transformations pour l'analyse des systèmes d'information. In *Actes du XXXVème Congrès INFORSID*, pages 93–110, 2017.

[ML16] Cedrine Madera and Anne Laurent. The next information architecture evolution : the data lake wave. In *Proceedings of the 8th International Conference on Management of Digital EcoSystems (MEDES'16)*, pages 174–180, 2016.

[MLNC17] Pierre Monnin, Mario Lezoche, Amedeo Napoli, and Adrien Coulet. Using formal concept analysis for checking the structure of an ontology in LOD : the example of dbpedia. In *Proceedings of the 23rd International Symposium on Foundations of Intelligent Systems (ISMIS'17)*, pages 674–683, 2017.

[MP07] Andreas Metzger and Klaus Pohl. Variability management in software product line engineering. In *Companion Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 186–187. IEEE Computer Society, 2007.

[MWC09] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof 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.

[MZB+17] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up technologies for reuse : automated extractive adoption of software product lines. In *Companion Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*, pages 67–70. IEEE Press, 2017.

[NBA+17] Sana Ben Nasr, Guillaume Bécan, Mathieu Acher, João Bosco Ferreira Filho, Nicolas Sannier, Benoit Baudry, and Jean-Marc Davril. Automated extraction of product comparison matrices from informal product descriptions. *Journal of Systems and Software*, 124 :82–103, 2017.

[NE09] Nan Niu and Steve M. Easterbrook. Concept analysis for product line requirements. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 137–148, 2009.

[NSKS06] Jyotirmaya Nanda, Timothy W Simpson, Soundar R Kumara, and Steven B Shooter. A methodology for product family ontology development using formal concept analysis and web ontology language. *Journal of computing and information science in engineering*, 6(2) :103–113, 2006.

[OSS04] Marek Obitko, Václav Snásel, and Jan Smid. Ontology design with formal concept analysis. In *Proceedings of the CLA 2004 International Workshop on Concept Lattices and their Applications (CLA'04)*, 2004.

[Ouz18] Amirouche Ouzerdine. Analyse formelle de données pour la sélection de plantes à effet pesticide pour la santé animale et végétale dans les pays du Sud. Master's thesis, Université de Montpellier, 2018.

[Par76] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on software engineering*, 2(1) :1–9, 1976.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software Product Line Engineering : Foundations, Principles, and Techniques*. Springer Science & Business Media, 2005.

[Pet01] Wiebke Petersen. A Set-Theoretical Approach for the Induction of Inheritance Hierarchies. *Electronic Notes in Theoretical Computer Science*, 53 :296–308, 2001.

[PEVD10] Jonas Poelmans, Paul Elzinga, Stijn Viaene, and Guido Dedene. Formal concept analysis in knowledge discovery : a survey. In *Proceedings of the 8th International Conference on Conceptual Structures (ICCS'10)*, pages 139–153, 2010.

[PRB11] Andreas Pleuss, Rick Rabiser, and Goetz Botterweck. Visualization techniques for application in interactive product configuration. In *Workshop Proceedings of the 15th International Software Product Line Conference (SPLC'11), Volume 2*, page 22, 2011.

[Pri05] Uta Priss. Linguistic applications of formal concept analysis. In *Formal Concept Analysis, Foundations and Applications*, pages 149–160, 2005.

[Pri06] Uta Priss. Formal concept analysis in information science. *Annual review of information science and technology*, 40(1) :521–543, 2006.

[PSA+12] Jeff Z Pan, Steffen Staab, Uwe Aßmann, Jürgen Ebert, and Yuting Zhao. *Ontology-driven software development*. Springer Science & Business Media, 2012.

[Rab09] Rick Rabiser. *A user-centered approach to product configuration in software product line engineering*. na, 2009.

[RATN17] Justine Reynaud, Mehwish Alam, Yannick Toussaint, and Amedeo Napoli. A Proposal for Classifying the Content of the Web of Data Based on FCA and Pattern Structures. In *Proceedings of 23rd International Symposium on Foundations (Methodologies) for Intelligent Systems (ISMIS'17)*, pages 684–694, 2017.

[RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities. In *Proceedings of the 6th World Conference on Integrated Design & Process Technology (IDPT'02)*, 2002.

[RHHNV13] Mohamed Rouane-Hacene, Marianne Huchard, Amedeo Napoli, and Petko Valtchev. Relational concept analysis : mining concept lattices from multi-relational data. *Annals of Mathematics and Artificial Intelligence*, 67(1) :81–108, 2013.

[RPK10] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Automatic variation-point identification in function-block-based models. In *ACM SIGPLAN Notices*, volume 46, pages 23–32. ACM, 2010.

[RPK11] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Extraction of feature models from formal contexts. In *Workshop Proceedings (Volume 2) of the 15th International Conference on Software Product Lines (SPLC'11)*, pages 4 :1–4 :8, 2011.

[RZK18] Luyao Ren, Shurui Zhou, and Christian Kästner. Forks insight : providing an overview of github forks. In *Proceedings of the 40th International Conference on Software Engineering : Companion Proceeedings (ICSE'18)*, pages 179–180, 2018.

[SAB13] Nicolas Sannier, Mathieu Acher, and Benoit Baudry. From comparison matrix to variability model : The wikipedia case study. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, pages 580–585, 2013.

[SBA+14] Nicolas Sannier, Guillaume Bécan, Mathieu Acher, Sana Ben Nasr, and Benoit Baudry. Comparing or configuring products : are we getting the right ones ? In *Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'14)*, pages 9 :1–9 :7, 2014.

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2) :456–479, 2007.

[SJ04] Klaus Schmid and Isabel John. A customizable approach to full lifecycle variability management. *Science of Computer Programming*, 53(3) :259–284, 2004.

[SLB+11] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 461–470, 2011.

[SMR08] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press, 2008.

[Sne00] Gregor Snelting. Software reengineering based on concept lattices. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 3–10, 2000.

[SOM+05] Malika Smaïl-Tabbone, Shazia Osman, Nizar Messai, Amedeo Napoli, and Marie-Dominique Devignes. BioRegistry : A Structured Metadata Repository for Bioinformatic Databases. In *Proceedings of the 1st International Symposium on Computational Life Sciences (CompLife'05)*, pages 46–56, 2005.

[SRA+14] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wasowski, and Krzysztof Czarnecki. Efficient synthesis of feature models. *Information & Software Technology*, 56(9) :1122–1143, 2014.

[SSD13] Hamzeh Eyal Salman, Abdelhak-Djamel Seriai, and Christophe Dony. Feature-to-code traceability in a collection of software variants : Combining formal concept analysis and information retrieval. In *Proceedings of the 14th IEEE International Conference on Information Reuse and Integration (IRI'13)*, pages 209–216, 2013.

[SSD14] Hamzeh Eyal Salman, Abdelhak Seriai, and Christophe Dony. Feature location in a collection of product variants : Combining information retrieval and hierarchical clustering. In *The 26th International Conference on Software Engineering and Knowledge Engineering (SEKE'14)*, pages 426–430, 2014.

[SSS17] Anas Shatnawi, Abdelhak-Djamel Seriai, and Houari A. Sahraoui. Recovering software product line architecture of a family of object-oriented product variants. *Journal of Systems and Software*, 131 :325–346, 2017.

[STB+02] Gerd Stumme, Rafik Taouil, Yves Bastide, Nicolas Pasquier, and Lotfi Lakhal. Computing iceberg concept lattices with Titanic. *Data Knowledge Engineering (DKE)*, 42(2) :189–222, 2002.

[TBD+08] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6) :883 – 896, 2008.

[TBK09] Thomas Thum, Don Batory, and Christian Kastner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 254–264, 2009.

[TCBE05] Thomas Tilley, Richard Cole, Peter Becker, and Peter W. Eklund. A Survey of Formal Concept Analysis Support for Software Engineering Activities. In *Formal Concept Analysis, Foundations and Applications*, volume 3626 of *Lecture Notes in Computer Science*, pages 250–271. Springer, 2005.

[TLYZ13] Lei Tan, Yuqing Lin, Huilin Ye, and Guoheng Zhang. Improving product configuration in software product line engineering. In *Proceedings of the 36th Australasian Computer Science Conference, Volume 135*, pages 125–133. Australian Computer Society, Inc., 2013.

[VdL02] Frank Van der Linden. Software product families in Europe : the Esaps & Café projects. *IEEE software*, 19(4) :41–49, 2002.

[VGBS01] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, pages 45–54, 2001.

[Wil02] Rudolf Wille. Why can concept lattices support knowledge discovery in databases ? *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2-3) :81–92, 2002.

[Wol01] Karl Erich Wolff. Temporal concept analysis. In *Proceedings of the 1s International Workshop on Concept Lattices-Based Theory, Methods and Tools for Knowledge Discovery in Databases, co-located with the 9th International Conference on Conceptual Structures (ICCS'01)*, pages 91–107, 2001.

[WR09] Ryen W. White and Resa A. Roth. *Exploratory Search : Beyond the Query-Response Paradigm*. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers, 2009.

[XXJ12] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Feature location in a collection of product variants. In *Proceedings of the 19th IEEE Working Conference on Reverse Engineering (WCRE'12)*, pages 145–154, 2012.