Composition des modèles de lignes de produits logiciels

Takoua Ben Rhouma Aouina

To cite this version:

Takoua Ben Rhouma Aouina. Composition des modèles de lignes de produits logiciels. Autre [cs.OH]. Université Paris Sud - Paris XI, 2012. Français. <NNT : 2012PA112299>. <tel-00772257>

HAL Id: tel-00772257
https://tel.archives-ouvertes.fr/tel-00772257

Submitted on 10 Jan 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ PARIS-SUD
ÉCOLE DOCTORALE D'INFORMATIQUE

THÈSE DE DOCTORAT

Présentée en vue d'obtenir le grade de docteur en informatique
par

Takoua BEN RHOUMA AOUINA

COMPOSITION DES MODÈLES DE LIGNES DE PRODUITS LOGICIELS

Soutenue le 29 novembre 2012
Devant la commission d'examen composée de :

Claude MARCHÉ,	Président,	INRIA-UPS
François TERRIER,	Directeur,	CEA LIST
Patrick TESSIER,	Encadrant,	CEA LIST
Camille SALINESI,	Rapporteur,	Université Paris 1 Panthéon-Sorbonne
Laurence DUCHIEN,	Rapporteur,	INRIA
Jean Claude ROYER,	Examinateur,	Ecole des mines-INRIA

Table des matières

1 Introduction 7
2 Etat de l'art 10
2.1. L'ingénierie des lignes de produits logiciels 11
2.1.1. Les lignes de produits logiciels 11
2.1.1.1. Apports de l'utilisation des lignes de produits logiciels 11
2.1.2. Environnement pour l'ingénierie des lignes de produits logiciels 13
2.1.2.1. L'ingénierie de domaine 14
2.1.2.2. L'ingénierie d'application 15
2.1.3. La variabilité : concept clé dans l'ingénierie des lignes de produits logiciels 16
2.2. Modélisation et composition des lignes de produits logiciels 17
2.2.1. Approches des modèles de caractéristiques 18
2.2.1.1. Modélisation 18
2.2.1.2. Composition 19
2.2.1.3. Synthèse 28
2.2.2. Approches de modélisation orientée aspects 29
2.2.2.1. Modélisation 29
2.2.2.2. Composition 29
2.2.2.3. Synthèse 31
2.2.3. Approches dirigées par les modèles annotatrices de variabilité 32
2.2.3.1. Modélisation 32
2.2.3.2. Composition 34
2.2.4. Bilan et positionnement 35
2.3. Conclusion 36
3 Choix de modélisation et aperçu global des contributions 38
3.1. Modèle de ligne de produits logiciels : vue de structure composite 39
3.2. Consolidation de modèles de lignes de produits logiciels 41
3. 2.1. Règle de connexion à un port simple (Reg1) 44
3.2.2. Règle de connexion à un port multiple (Reg2) 45
3.2.3. Règle de port variable attaché sur une part variable (Reg3) 47
3.2.4. Règle de port variable connecté à un port attaché sur une part variable (Reg4) 48
3.2.5. Règle de connecteur variable reliant deux ports simples (Reg5) 50

3.2.6. Règle de connecteur variable reliant deux ports dont au moins un est multiple (Reg6) 51
3.2.7. Synthèse 55
3.3. Formes de composition 55
3.4. Conclusion 58
4 Fusion des modèles de lignes de produits logiciels 59
4.1. Exemple illustratif 60
4.2. Démarche de fusion : aperçu global 62
4 .3. Consolidation des modèles en entrée : première phase de fusion 63
4.4. Fusion des modèles en entrée : deuxième phase de fusion 65
4.4.1. Etape de comparaison 67
4.4.2. Etape de fusion 71
4.4.2.1. Propriétés sémantiques de fusion 71
4.4.2.2. Fusion des éléments structurels 72
4.4.2.3. Fusion des contraintes de variabilité 78
4.4.3. Synthèse 82
4.5. Consolidation du modèle de fusion résultant : troisième phase de fusion 83
4.6. Conclusion 85
5 Agrégation des modèles de lignes de produits logiciels 86
5.1. Exemple illustratif 87
5.2. Démarche d'agrégation : aperçu global 88
5.3. Consolidation des modèles en entrée : première phase d'agrégation 90
5.4. Agrégation des modèles en entrée : deuxième phase d'agrégation 93
5.4.1. Propriété sémantique de l'agrégation 96
5.4.2. Proposition1 97
5.4.2.1. Règles d'agrégation 97
5.4.2.2. Synthèse 103
5.4.3. Proposition2 103
5.4.3.1 Règles d'agrégation : 103
5.4.3.2. Synthèse 109
5.4.4. Agrégation et effets de bord 110
5.5. Conclusion 111
6 Evaluation 113
6.1. SEQUIOA : Environnement pour le développement des lignes de produits logiciels 114

6.1.1. Processus de développement SEQUOIA 114
6.1.2. Architecture 115
6.1.2.1. Outil de propagation 116
6.1.2.3 Outil de calcul de produits 117
6.1.2.4. Outil de génération de modèle de décision 117
6.1.2.5. Outil de dérivation 118
6.2. Implémentation 119
6.2.1. Outil de consolidation 119
6.2.2. Outil de fusion 122
6.3. Consolidation de modèles de lignes de produits logiciels : Evaluation et apports 124
6.3.1. Stratégie d'évaluation 124
6.3.2. Résultats 125
6.4. Fusion de modèles : Evaluation et apports 128
6.4.1. Stratégie d'évaluation 128
6.4.2. Résultats 128
6.5. Conclusion 129
7 Conclusion 131
Expression de la variabilité dans SEQUOIA 137
Bibliographie 147

Introduction

La réutilisation s'impose de plus en plus comme une solution méthodologique et technologique pour le développement de systèmes logiciels complexes. Un des principaux défis de la communauté de recherche et des industries est de trouver les concepts, mécanismes et langages adéquats pour une meilleure réutilisation et configuration en masse des systèmes logiciels.

Par ailleurs, l'ingénierie des lignes de produits logiciels est un paradigme qui prône une vision de modélisation et de développement dans laquelle l'objectif envisagé n'est plus l'obtention d'un système logiciel, mais d'un ensemble de systèmes logiciels possédant des caractéristiques communes. L'utilisation des lignes de produits logiciels s'appuie principalement sur la réutilisation et la configuration en masse.

La modélisation est une phase cruciale dans le développement des lignes de produits logiciels. Elle permet de décrire les similarités et les différences dans une famille de systèmes ou de produits logiciels. Plusieurs approches basées sur les modèles proposent des mécanismes pour le développement et la maintenance des lignes de produits logiciels. Nous citons à titre d'exemples les approches basées sur les modèles de caractéristiques [1][2] ainsi que les approches basées sur les modèles UML (Unified Modeling Language) [3][4].

La modélisation, cependant, peut se révéler une tâche difficile voir infaisable quand il s'agit de modéliser des lignes de produits logiciels complexes [5]. En effet, le grand nombre de caractéristiques et de composants dans un modèle de ligne de produits logiciels dépasse la capacité du concepteur à les comprendre en détails. De plus, les lignes de produits logiciels sont sujettes à évolution puisqu'elles doivent satisfaire l'évolution continue des attentes des clients ainsi que les progrès technologiques.

En pratique, la tâche de modélisation est distribuée sur différents intervenants ou équipes. Elle permet ainsi d'améliorer la flexibilité et la compréhension du système en réduisant le temps de développement [6][7]. La distribution de la tâche de modélisation peut être réalisée par exemple selon les préoccupations [8][9] ou aussi selon les catégories des clients visés [10]. Les modèles développés par les différents intervenants sont des fragments qui représentent des vues partielles du modèle global de la ligne de produits logiciels. Pour finalement obtenir le modèle global de la ligne de produits logiciels, les modèles développés séparément doivent alors être composés.

La composition des modèles de lignes de produits logiciels est une tâche complexe. En effet, la taille des modèles est souvent large. Composer ces modèles demande de savoir comment naviguer à travers leurs différents éléments et manipuler ces derniers durant le processus de composition. Ce travail devient plus fastidieux et compliqué à réaliser dans le cas de modèles non triviaux possédant une structure complexe. D'autre part la variabilité est une notion clé dans le contexte des lignes de produits logiciels. Tous les éléments appartenant à un modèle de ligne de produits logiciels possèdent une information de variabilité qui indique les possibilités de leurs présences dans les produits logiciels envisagés. De ce fait, calculer l'information de variabilité des éléments résultants dans le modèle global de la ligne de produits logiciels est un des principaux points clés à résoudre durant le processus de composition. En plus de l'information de variabilité, les contraintes de variabilité associées aux différents éléments variables sont d'une importance majeure. Ces contraintes spécifient les produits logiciels valides d'une ligne de produits logiciels. Le processus de composition des modèles de lignes de produits logiciels doit donc combiner les contraintes de variabilité des modèles en entrée et en déduire les contraintes de variabilité associées au modèle global de la ligne de produits logiciels. Enfin, la composition des modèles de lignes de produits logiciels doit avoir une sémantique bien précise afin de cerner l'objectif qu'elle vise. Un nombre de critères doit donc être vérifié par le résultat obtenu pour satisfaire la sémantique de la composition.

Deux points de vue sont considérés dans la modélisation des lignes de produits logiciel. D'une part l'architecture pour laquelle UML est de plus en plus utilisé et d'autre part la variabilité qui s'appuie parfois sur un formalisme orthogonal. Il devient alors nécessaire de réconcilier ces deux points de vue afin d'exploiter la ligne de produits logiciels.

L'approche contenue dans cette thèse consiste à coupler les deux points de vue en annotant les modèles d'architecture UML avec l'information de variabilité afin d'assurer la cohérence de l'ensemble. C'est ainsi que se pose le problème de composition qui doit assurer la combinaison des modèles UML annotés par la variabilité.

Des mécanismes de composition spécifiques aux modèles de lignes de produits logiciels représentés par des modèles UML et incluant des annotations de variabilité doivent alors être fournis. Ces mécanismes doivent prendre en considération la structure des modèles à composer, l'information de variabilité des éléments structurels ainsi que les contraintes de variabilité qui leurs sont associées. Pour aborder un tel problème, émergeant principalement de l'association de la variabilité avec la structure des modèles, nous proposons de le résoudre en deux volets ; une première méthodologie de composition, consacrée à la fusion, permet de

combiner des modèles représentant des similarités au niveau des éléments structurels. Une seconde méthodologie de composition, nommée agrégation, permet de combiner des modèles n'ayant pas de similarités et dont les éléments structurels peuvent être liés par des contraintes transversales.
Notre contribution assiste le concepteur dans la modélisation et la composition des grands modèles de lignes de produits logiciels en considérant l'aspect structurel, la variabilité des éléments et les contraintes qui leurs sont associées. La représentation sous forme de structures composites d'UML favorise la réutilisation de ces modèles.

Ce rapport est structuré selon le plan suivant :

- Le deuxième chapitre classifie les approches de modélisation et de composition des lignes de produis logiciels. Les approches de composition sont alors analysées afin de positionner les contributions de la thèse par rapport aux méthodes de composition existantes.
- Le troisième chapitre consiste à présenter le choix de modélisation des lignes de produits logiciels. Le besoin de consolider les modèles est mis en avant et un ensemble de règles de consolidation sont alors proposées pour cette raison. Un aperçu global est donné ensuite sur les deux mécanismes proposés pour la composition des modèles de lignes de produits logiciels : la fusion et l'agrégation.
- Le quatrième chapitre présente la démarche proposée pour la fusion des modèles de lignes de produits logiciels. Il détaille et argumente les différentes phases de fusion.
- Le cinquième chapitre présente la démarche proposée pour l'agrégation des modèles de lignes de produits logiciels. Les différentes phases d'agrégation sont détaillées et argumentées.
- Le sixième chapitre évalue la monté en échelle du mécanisme de fusion. L'évaluation des règles de consolidation montre comment le nombre de produits incomplets éliminés augmente de manière proportionnelle au nombre d'éléments variables dans le modèle de ligne de produits logiciels.

Enfin nous concluons la thèse par une présentation des apports, des limitations et des perspectives de ce travail de recherche.

CHAPITRE 2

Etat de l'art

2.1. L'ingénierie des lignes de produits logiciels 11
2.1.1. Les lignes de produits logiciels 11
2.1.1.1. Apports de l'utilisation des lignes de produits logiciels 11
2.1.2. Environnement pour l'ingénierie des lignes de produits logiciels 13
2.1.2.1. L'ingénierie de domaine 14
2.1.2.2. L'ingénierie d'application 15
2.1.3. La variabilité : concept clé dans l'ingénierie des lignes de produits logiciels 16
2.2. Modélisation et composition des lignes de produits logiciels 17
2.2.1. Approches des modèles de caractéristiques 18
2.2.1.1. Modélisation 18
2.2.1.2. Composition 19
2.2.1.3. Synthèse 28
2.2.2. Approches de modélisation orientée aspects 29
2.2.2.1. Modélisation 29
2.2.2.2. Composition 29
2.2.2.3. Synthèse 31
2.2.3. Approches dirigées par les modèles annotatrices de variabilité 32
2.2.3.1. Modélisation 32
2.2.3.2. Composition 34
2.2.4. Bilan et positionnement 35
2.3. Conclusion 36

L'objectif de ce chapitre est de positionner les contributions de cette thèse par rapport aux travaux existants. Pour ce faire, il est structuré en deux sections. La première section présente l'ingénierie des lignes de produits logiciels et ses principaux apports et activités. La deuxième section analyse les approches de composition les plus pertinentes pour ce travail. Elle propose de regrouper les approches selon le type de modélisation qu'elles utilisent pour représenter les lignes de produits logiciels. Pour chaque type de modélisation, les concepts sont présentés. Les approches proposées pour la composition de ces modèles sont analysées par rapport à la gestion de l'aspect structurel et de la consistance des modèles, l'information de variabilité des éléments, les contraintes de variabilité et la sémantique de composition visée.

2.1. L'ingénierie des lignes de produits logiciels

2.1.1. Les lignes de produits logiciels

L'ingénierie des lignes de produits logiciels est une méthodologie qui permet le développement d'une multitude de produits ou systèmes logiciels avec un gain considérable en termes de coût, de temps et de qualité [12][13][14]. Il ne s'agit plus de modéliser et de développer des systèmes individuels. L'ingénierie des lignes de produits logiciels s'intéresse plutôt à modéliser et développer une famille de produits. Une ligne de produits logiciels consiste donc en un ensemble de produits représentant des similarités. La définition des lignes de produits logiciels proposée par Clements et Northrop donne une idée plus précise sur leur principe [13]:

"Une ligne de produits logiciels est un ensemble de produits logiciels qui partagent et gèrent un ensemble de caractéristiques satisfaisant les besoins spécifiques d'un segment de marché particulier ou une mission et qui sont développées à partir d'un même ensemble d'atouts essentiels décrits d'une manière prescrite"

L'utilisation des lignes de produits logiciels permet de bénéficier de plusieurs avantages. Nous spécifions dans la sous-section suivante certains de ses avantages.

2.1.1.1. Apports de l'utilisation des lignes de produits logiciels

L'utilisation des lignes de produits logiciels s'appuie sur la réutilisation ainsi que la configuration en masse des systèmes logiciels. La réutilisation a été définie par Krueger comme [15]:

"Le processus de création des systèmes logiciels à partir des logiciels existants au lieu de les créer à partir de zéro"

La réutilisation n’est pas une idée nouvellement utilisée dans la production des systèmes logiciels. Elle a déjà été utilisée comme stratégie de production à des fins précises comme la diminution des coûts de développement et l’amélioration de la qualité [16][17]. Les anciennes stratégies de réutilisation s’intéressaient plutôt à la réutilisation des pièces de codes relativement petites comme par exemple les librairies qui contiennent des algorithmes, modules, objets ou composants. Le défit rencontré à ce niveau consiste à localiser les pièces à réutiliser et à les intégrer avec un effort inférieur à celui de leur création à partir de zéro. Cependant, ce compromis reste difficile à réaliser. L’effort de réutilisation demeure souvent plus important que la création à partir de zéro [14]. L’idée parvenue du concept des lignes de produits a ensuite offert plus de facilité pour la réutilisation puisqu’il ne s’agit plus de construire des systèmes uniques au cas par cas mais plutôt de viser dès le début une famille de systèmes qui appartiennent au même domaine et qui représentent des similarités [18]:

"La réutilisation des pièces larges fonctionne mieux dans les familles de systèmes reliés, et donc dépendants du même domaine"

Les retours d’expériences montrent qu’en absence de planification efficace, le coût de réutilisation s’avère supérieur au coût de la construction à partir de zéro [14]. Ce problème est résolu pour les approches de développement des lignes de produits logiciels ; la réutilisation est planifiée, permise et applicable [13]. Les éléments dont le développement à partir de zéro est le plus coûteux comme par exemple les exigences, les structures et les composants sont conçus pour être réutilisés et optimisés dans l’optique de les utiliser dans le développement de plusieurs systèmes.

En plus de la réutilisation, la configuration de masse est reconnue comme une des principales caractéristiques des lignes de produits logiciels [14]. Elle a été définie par Davis comme [19]:

"La production à grande échelle de biens adaptés aux besoins individuels des clients "

L’ingénierie des lignes de produits logiciels s’appuie ainsi sur la configuration en masse des produits logiciels en se basant sur les éléments communs qui sont créés une fois et qui sont réutilisés dans tous les systèmes de la ligne de produits. Les éléments optionnels, quant à eux, ils sont variés pour obtenir à la fin plusieurs produits où chacun est adapté aux besoins d’une catégorie spécifique d’utilisateurs. Par exemple dans l’industrie de l’automobile, certains véhicules sont conçus pour le déplacement d’une seule personne alors que d’autres sont conçus pour le déplacement de plusieurs personnes. Ainsi, l’ingénierie des lignes de produits logiciels renforce la production en masse des systèmes logiciels adaptés aux besoins des utilisateurs.

Dès lors, le temps de mise sur le marché et les coûts de développement sont réduits pendant que la qualité des produits s'améliore [12][13][14].

La figure 2.1 montre le gain de temps pour la mise sur le marché avec et sans utilisation de l'ingénierie des lignes de produits. Le temps de mise sur le marché pour un produit unique reste constant. Pour l'ingénierie des lignes de produits, le temps de construction des éléments communs à tous les produits est important. Il est cependant compensé par la réutilisation des éléments communs.

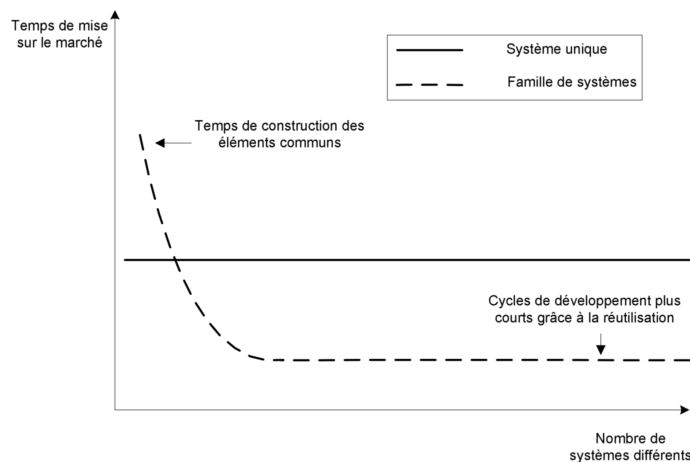


Figure 2.1 : temps de mise sur le marché avec et sans l'ingénierie de lignes de produits [14]

Les succès dus à l'utilisation de l'ingénierie des lignes de produits sont nombreux et touchent plusieurs domaines comme l'industrie de l'automobile, les systèmes avioniques et la téléphonie. Citons à titre d'exemple Nokia qui arrive à produire de 25 à 30 modèles de téléphones différents par an grâce à l'utilisation du concept de lignes de produits [13].

2.1.2. Environnement pour l'ingénierie des lignes de produits logiciels

L'ingénierie des lignes de produits logiciels est décomposée en deux phases complémentaires [14]: l'ingénierie de domaine et l'ingénierie d'application (voir figure 2.2). L'ingénierie de domaine adopte la stratégie de développement pour la réutilisation, alors que l'ingénierie d'application s'appuie sur la stratégie du développement avec réutilisation. Le principe derrière ces deux phases rejoint l'idée de compenser le temps de développement des artéfacts communs, pendant la première phase, par la réutilisation de ces artéfacts pendant la deuxième phase.

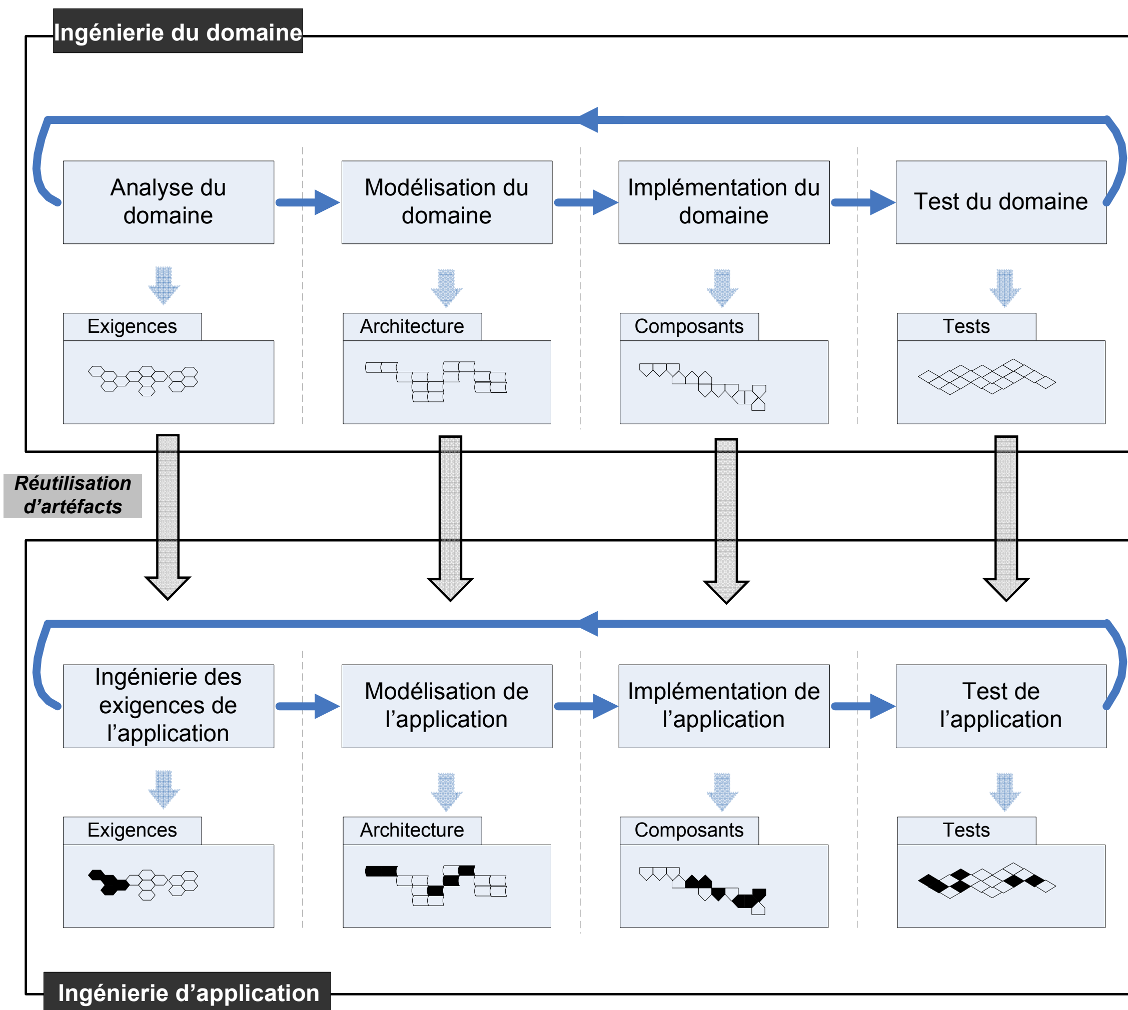


Figure 2.2 : Ingénierie de domaine et ingénierie d'application [14]

Nous décrivons dans la suite les deux phases, ingénierie de domaine et ingénierie d'application, illustrées dans la figure 2.2.

2.1.2.1. L'ingénierie de domaine

Cette phase permet de gérer une ligne de produits logiciels dans sa globalité et non pas comme un ensemble de produits séparés où chaque produit est traité indépendamment des autres. La ligne de produits logiciels doit alors inclure les besoins spécifiés par toutes les catégories d'utilisateurs visées.

Dés lors, il est possible de définir la portée de la ligne de produits logiciels, c'est-à-dire l'ensemble des produits planifiés. La définition des membres de la famille de produits planifiée permet d'identifier et de planifier l'implémentation des points communs réutilisables ainsi que les points de différence entre eux. C'est ainsi que cette phase adopte la stratégie de développement pour la réutilisation. En effet, les artéfacts logiciels communs définis, comme

par exemple les exigences, les composants et les classes, sont construits de façon à ce qu'ils soient réutilisés dans les produits planifiés.

Pour accomplir ces objectifs, l'ingénierie de domaine est composée de quatre activités qui sont l'analyse de domaine, la modélisation de domaine, l'implémentation de domaine et le test de domaine.

- Analyse de domaine : l'analyse de domaine inclut l'ingénierie des exigences du domaine. En effet, les exigences sont analysées pour identifier celles qui sont communes à tous les produits et celles qui sont spécifiques à des produits bien particuliers.
- Modélisation de domaine : cette activité s'intéresse à la définition de l'architecture de la ligne de produits logiciels. Cette architecture donne une vue structurelle sur les différents membres de la ligne de produits logiciels. Cette activité doit alors inclure les mécanismes de variabilité utilisés et identifier les parties réutilisables de l'architecture.
- Réalisation (implémentation) de domaine : les différentes parties réutilisables de l'architecture sont détaillées et implémentées dans des composants logiciels réutilisables.
- Test de domaine : cette activité est responsable de la validation et la vérification des composants réutilisables obtenus à l'issue de l'activité précédente. Les tests effectués sur les composants se font par rapport à leur conformité aux spécifications données dans les exigences, l'architecture et les artéfacts de modèle.

2.1.2.2. L'ingénierie d'application

Cette phase permet d'obtenir les produits concrets à partir des artéfacts identifiés dans la phase d'ingénierie de domaine. La stratégie adoptée à ce niveau se base sur le développement avec réutilisation. En effet, cette phase a comme but de maximiser le taux de réutilisation des artéfacts définis et développés pour un produit bien spécifique de la ligne de produits. Les points communs et les points de différence doivent ainsi être exploités pour la construction des produits. La construction de ces produits est appelée dérivation de produits. Elle consiste à sélectionner les artéfacts qui seront présents dans un produit donné. Comme pour l'ingénierie de domaine, l'ingénierie d'application est composée de quatre activités à savoir ingénierie des exigences d'application, la modélisation d'application, l'implémentation d'application et le test d'application.

- Ingénierie des exigences de l'application : elle s'intéresse à désigner les exigences spécifiques aux produits souhaités. Ces exigences ont un impact direct sur les artéfacts du domaine qui seront réutilisés dans la construction du produit.

- Modélisation de l’application : elle permet d’obtenir l’architecture du produit à partir de l’architecture globale définie pendant la modélisation du domaine. Pour ce faire, les parties requises du modèle sont sélectionnées et incorporées.
- Réalisation (implémentation) de l’application : cette activité consiste à la réalisation du produit concret souhaité. Les composants qui correspondent aux parties sélectionnées dans l’étape précédente sont alors assemblés.
- Test de l’application : cette activité valide et vérifie que le produit obtenu à l’issue de l’activité précédente est conforme aux exigences, à l’architecture et aux parties du modèle sélectionnés.

La modélisation des lignes de produits facilite ainsi la réutilisation des concepts manipulés. Plusieurs approches dirigées par les modèles s’intéressent au développement et à la maintenance des lignes de produits logiciels. Cependant, la modélisation demeure une tâche complexe voir irréalisable pour les lignes de produits logiciels complexes et à très grande échelle [20]. C’est ainsi qu’est venue l’idée de distribuer la tâche de modélisation sur plusieurs équipes, chacune travaille sur un modèle partiel. Chaque fragment de modèle représente une vue partielle du modèle global de la ligne de produits qui doit être obtenu par composition des fragments de modèles qui le constituent.

La composition de modèles de lignes de produits logiciels renforce la réutilisation et permet d’obtenir de nouveaux modèles de lignes de produits logiciels. Cependant cette tâche est loin d’être facile ou évidente, d’autant plus qu’il s’agit de modèles intégrant des éléments variables. La variabilité est donc un point principal à gérer durant la composition de ces modèles, d’où l’intérêt d’investir dans des travaux de recherche à ce sujet [5]. La sous section suivante présente le rôle joué par la variabilité dans le contexte des lignes de produits logiciels.

2.1.3. La variabilité : concept clé dans l’ingénierie des lignes de produits logiciels

Dans le contexte des lignes de produits logiciels, la notion de variabilité permet le développement des produits configurés par réutilisation des artéfacts prédéfinis et ajustables. Elle est modélisée au niveau de la phase d’ingénierie de domaine pour représentés les différences entre les produits planifiés.

Plusieurs définitions on été attribuées à la variabilité. Par exemple, Weiss et Lai définissent la variabilité comme *"une hypothèse sur la façon dont les membres d’une famille peuvent se différencier entre eux"* [12]. Ceci explique le rôle que joue la variabilité dans l’adaptation des

produits aux besoins spécifiques des catégories d'utilisateurs visées, pendant que les besoins communs sont présents et partagés par tous les membres de la ligne de produits.
La variabilité a aussi été classée en deux types : la variabilité temporelle et la variabilité spatiale. Pohl et al. ont défini la variabilité temporelle comme *"l'existence de différentes versions d'un artéfact qui sont valides à des moments différents"* et la variabilité spatiale comme *"l'existence d'un artéfact sous différentes formes en même temps"* [14]. Cependant, la variabilité spatiale reste la plus traitée dans la pratique de l'ingénierie des lignes de produits logiciels. La variabilité représente ainsi un concept clé dans l'ingénierie des lignes de produits logiciels et doit être gérée tout au long du processus de développement. La composition des modèles de lignes de produits logiciels doit alors prendre en compte la variabilité des éléments de modèles et la gérer. Ceci complique d'autant plus le mécanisme de composition.
La section suivante donne un aperçu sur les modèles utilisés pour représenter les lignes de produits logiciels. Elle se focalise sur les mécanismes de composition fournis et analyse leurs propositions pour gérer la variabilité durant la composition. D'autres critères d'analyse sont aussi considérés à savoir la gestion des structures de modèles, leurs consistances, les contraintes de variabilité ainsi que la sémantique de la composition.

2.2. Modélisation et composition des lignes de produits logiciels

Pendant le développement des lignes de produits logiciels, le concepteur se trouve en face d'une multitude d'éléments à gérer. Pour faciliter la manipulation de ces éléments, il est important de les modéliser. Il s'agit de représenter une abstraction des éléments appartenant à la ligne de produits étudiée pour diminuer leur complexité. Kramer a définit le but de l'abstraction par [5]:
"la traçabilité des correspondances à partir d'une représentation d'un problème vers une autre représentation qui préserve certaines propriétés souhaitées et qui réduit la complexité"
Dans le contexte des lignes de produits logiciels, un modèle représente une abstraction de la ligne de produits logiciels considérée. Dans cette section, nous distinguons trois catégories d'approches pour la modélisation et la composition des lignes de produits logiciels. Nous présentons pour chaque catégorie, une brève description des modèles utilisés pour la représentation des lignes de produits logiciels ainsi que les approches qui les adoptent. Ensuite une analyse détaillée est présentée pour les mécanismes de composition proposés dans chacune des trois catégories. Cette analyse présente certains travaux pertinents et discute leurs avantages et inconvénients durant la composition des modèles de lignes de produits logiciels.

2.2.1. Approches des modèles de caractéristiques

2.2.1.1. Modélisation

Commençons d'abord par définir ce qu'est une caractéristique ou *Feature*. Dans la littérature, plusieurs définitions ont été données au concept de *Feature* [1][26][27][2][28][29][30][31][32][33]. Nous citons par exemple la définition donnée par Kang [1] qui considère qu’une *Feature* est :

"tout aspect important et distinctif ou caractéristique visible par les diverses parties prenantes", comme l'utilisateur final, l'expert du domaine, etc…

Les diagrammes de caractéristiques fournissent une représentation explicite et concise de la variabilité. Ils représentent les choix à faire pour déterminer les caractéristiques des produits envisagés [1][2]. Il s’agit d’une description graphique sous forme d’une hiérarchie des exigences d’une ligne de produits logiciels.

Plusieurs approches ont proposé de modéliser les lignes de produits logiciels par des modèles de caractéristiques à commencer par l’approche FODA (Feature Oriented Domain Analysis) de Kang [1]. Cette approche a pour objectif de capturer les points communs et les points de différences au niveau exigence. La caractéristique racine représente la ligne de produits logiciels, comme la caractéristique *Security* dans l’exemple de la figure 2.3. Cette caractéristique est décomposée en sous-caractéristiques qui sont elles aussi décomposées jusqu’à atteindre les caractéristiques feuilles. Des symboles graphiques spécifiques permettent de distinguer les caractéristiques obligatoires de celles optionnelles. Comme le montre l’exemple de la figure 2.3, la caractéristique *OutsideDetection* est obligatoire tandis que la caractéristique *Alarm* est optionnelle. Les relations entre les caractéristiques sont elles aussi représentées par des symboles graphiques comme par exemple la relation *or* représentée dans la figure pour spécifier que la caractéristique *Sensing* peut être réalisée via un détecteur infra-rouge *InfraredDetector,* un détecteur volumétrique *VolumetricDetector* ou bien les deux ensemble. L’alarme par contre peut être silencieuse, sous forme de sirène ou bien visuelle mais pas les trois ensemble. D’autre contraintes dites transversales de types inclusion, exclusion sont considérée par l’approche. Les contraintes d’une ligne de produits définissent les produits logiciels valides qui peuvent être obtenus.

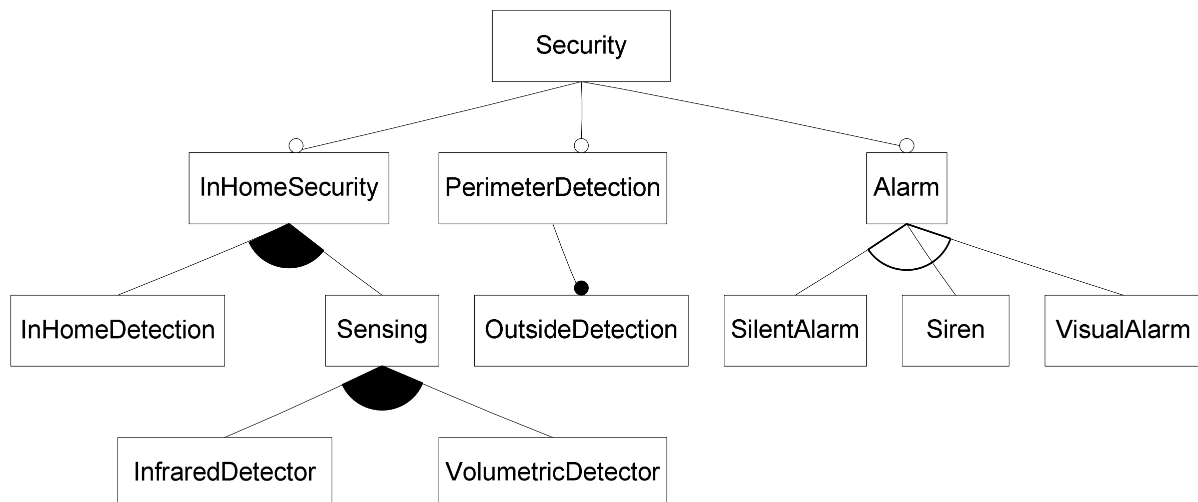


Figure 2.3 : Modèle de caractéristiques

Plusieurs autres approches ont succédé l'approche FODA et ont permis son extension. Par exemple l'approche FORM (Feature Oriented Reuse Method) [27], dont le principal apport est la décomposition du modèle de caractéristiques en couches permettant de décrire différents points de vue (exemple : capacité, environnement, technologie) concernant le développement de produits. L'approche FeatureRSEB [34][35][36] traite la cohérence entre le modèle de caractéristiques et les modèles de conception tels que le modèle de cas d'utilisation. L'approche Generative Programming [2] a pour objectif l'automatisation des phases de développement des lignes de produits et la génération automatique de code.

2.2.1.2. Composition

Plusieurs travaux, relevés dans la littérature, se sont focalisés sur la composition des modèles de caractéristiques [38][39][40][41][42][43]. Parmi cette liste non exhaustive, nous pouvons retenir :

Approche à base de règles de transformation de modèles

Alves et al. [38] ont motivé le besoin de faire évoluer les lignes de produits logiciels et plus particulièrement les modèles de caractéristiques. Dans ce travail, les auteurs proposent d'étendre la notion traditionnelle de transformation pour les lignes de produits logiciels. La transformation de modèles est satisfaite lorsque l'ensemble des systèmes valides du modèle de caractéristiques reste le même ou évolue. Pour ce faire, ils proposent un ensemble de règles de transformation qui considèrent les spécificités du modèle de caractéristiques. La figure 2.4 présente un exemple de règle de transformation. Cette règle prend en entrée un modèle de

caractéristiques incluant la caractéristique variable B et les caractéristiques C et D reliés par une contrainte de type ***xor***. Ceci veut dire que les systèmes valides sont ABC, AC, ABD et AD. La transformation de ce modèle donne un modèle de caractéristiques où les caractéristiques B, C et D sont reliés par une contrainte de type ***or***. L’ensemble des systèmes obtenus à partir de ce modèle inclut l’ensemble des systèmes valides obtenus à partir du premier modèle.

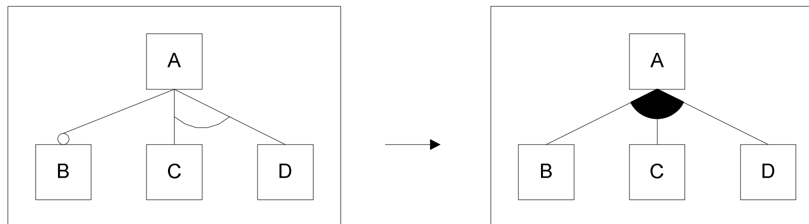


Figure 2.4 : Règle de transformation de variable et xor en or

La proposition présentée dans ce travail ne se limite pas à la transformation des modèles de caractéristiques. Les auteurs suggèrent aussi l’utilisation des règles de transformation dans la fusion des modèles de caractéristiques. Cependant, les règles de transformation proposées ont aussi besoin d’être étendues pour traiter les différents cas rencontrés dans la fusion. Ces règles traitent principalement les contraintes de variabilité sans intérêt particulier à la structure des caractéristiques ou leurs informations de variabilité. De plus, les règles proposées se limitent à gérer des contraintes de types *and*, *or* et *xor*. Les contraintes transversales de types *implication* et *équivalence* ne sont pas traitées par les règles proposées, malgré leur importance et la fréquence de leur utilisation dans le contexte des lignes de produits. Les règles de fusion de contraintes ne sont pas génériques, elles restent limitées aux types de contraintes spécifiées. Des précisions supplémentaires sont aussi nécessaires pour expliquer les critères de comparaison des caractéristiques à fusionner ainsi que les propriétés sémantiques de la fusion.

Approche à base de règles visuelles de fusion de modèles

Inspirés par le travail d’Alves et al., Segura et al. [39] proposent un catalogue de règles visuelles pour la fusion des modèles de caractéristiques. Chaque règle est composée de deux parties ; la partie gauche représente les deux patrons des modèles de caractéristiques à fusionner en entrée (les pré-conditions). La partie droite représente le patron du modèle de

caractéristiques résultant de la fusion (post-conditions). La figure 2.5 donne un exemple de ces règles.

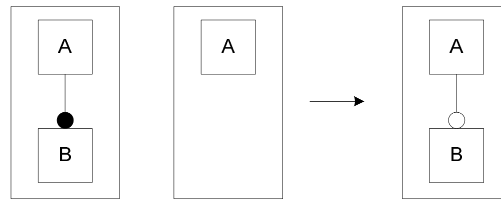


Figure 2.5 : Règle de fusion des caractéristiques

Les règles de fusion proposées permettent d’obtenir un modèle de caractéristiques dont l’ensemble des systèmes valides inclut au moins l’ensemble des systèmes valides obtenus à partir des modèles de caractéristiques en entrée. Ceci implique que le modèle de caractéristiques résultant permet d’obtenir des produits qui ne sont valides pour aucun des modèles d’entrée. La sémantique de la fusion manque de précisions pour les règles proposées. En effet, pour une sémantique donnée de la fusion (par exemple en mode union ou en mode intersection), le catalogue de règles doit être trié ; certaines règles sont maintenues pour la fusion alors que d’autres doivent être mises à jour selon la propriété sémantique choisie.

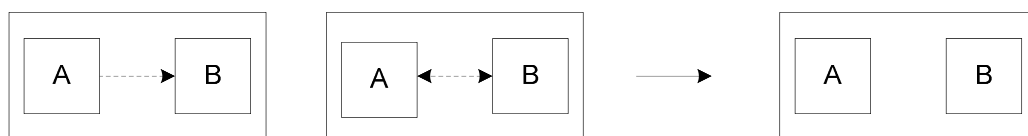


Figure 2.6 : Règle de fusion des caractéristiques transversales

La règle présentée dans la figure 2.6, montre la fusion de deux patrons de modèles de caractéristiques. Le premier contient deux caractéristiques A et B ayant une contrainte associée de type *implication*. Le deuxième inclut deux caractéristiques A et B ayant une contrainte associée de type *équivalence*. Le résultat de leur fusion donne un modèle de caractéristiques qui inclut les deux caractéristiques variables A et B permettant ainsi toutes les combinaisons possibles de A et B. Pour une fusion en mode union par exemple, cette contrainte permet d’obtenir des systèmes valides qui n’appartiennent à aucun des modèles d’entrée. Le système qui contient la caractéristique A est un système valide obtenu à partir du modèle de fusion. Cependant, ce système n’est valide pour aucun des modèles de

caractéristiques en entrée. Cette règle doit alors être mise à jour pour satisfaire la propriété union de la fusion.

Les auteurs proposent des règles de fusion pour spécifier l'information de variabilité des éléments résultant de la fusion, comme l'exemple de la règle illustrée dans la figure 2.5. Ils proposent aussi un catalogue de règles pour la fusion des contraintes, comme l'exemple dans la figure 2.6. Cependant, l'énumération de ces règles selon les types de contraintes n'offre pas de généricité et limite la fusion aux types de contraintes identifiées. De plus, le catalogue de règles contient 30 règles qui doivent être vérifiées selon la sémantique de fusion.

Approche à base d'opérateurs de fusion et d'agrégation de modèles

Acher et al. proposent [42] un ensemble d'opérateurs pour la composition de modèles de caractéristiques. Nous distinguons plus particulièrement les deux opérateurs suivants :

- ***Opérateur de fusion :*** cet opérateur permet de fusionner des parties communes de modèles de caractéristiques pour obtenir un modèle de caractéristiques intègre. L'opérateur de fusion utilise le nom des caractéristiques comme critère de correspondance. Les caractéristiques ayant le même nom sont alors fusionnées. Cependant, ce critère reste étroitement lié au contexte de modèle de caractéristiques. Il ne présente pas de problèmes de fusion pour ce type de modèles qui est généralement utilisé pour représenter les lignes de produits à un niveau d'abstraction élevé. Ceci est loin d'être similaire pour des modèles de nature différente comme par exemple les modèles UML et qui représentent un niveau d'abstraction moins élevé que les modèles de caractéristiques. Comparer les éléments de modèles selon un seul critère qui est leurs noms est relativement facile à implémenter comme critère de correspondance. Cependant, ce critère peut être très permissif dans certains cas. Par exemple, comparer les attributs de classes en se basant sur leurs noms comme critère de correspondance peut impliquer des problèmes lors de la fusion si les types associés à ces attributs sont incompatibles.

La figure 2.7 montre l'exemple des caractéristiques *Sensing* appartenant à deux modèles de caractéristiques différents. Selon la proposition d'Acher, deux éléments des modèles peuvent être fusionnés s'ils ont le même nom. Nous déduisons donc que *Sensing* de modèle1 correspond à *Sensing* de modèle2. De même, *infraredDetector* et *volumetricDetector* de modèle1 correspondent respectivement à *infraredDetector* et *volumetricDetector* de modèle2. Par conséquent les éléments *Sensing*, *infraredDetector* et *volumetricDetector* de modèle1 peuvent être fusionnés avec les éléments respectifs *Transport*, *infraredDetector* et *volumetricDetector* de modèle2.

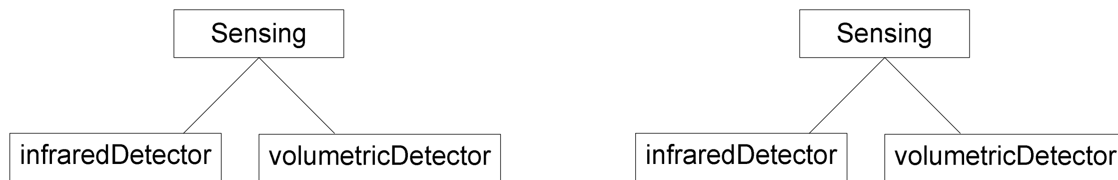


Figure 2.7 : Exemple de correspondance des caractéristiques *Sensing*

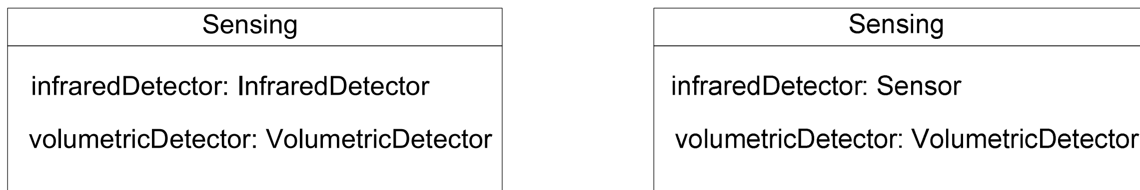


Figure 2.8 : Exemple de correspondance des classes *Sensing*

Nous avons ensuite représenté les mêmes éléments dans un modèle UML illustré dans la figure 2.8. En utilisant le même critère de comparaison, nous constatons que les classes *Sensing* de modèle1 et modèle2 correspondent (même nom). Les attributs *volumetricDetector* de modèle1 et modèle2 correspondent aussi et possèdent des types compatibles, contrairement aux attributs *infraredDetector* qui correspondent selon le critère du nom mais qui ont des types incompatibles (*InfraredDetector* et *Sensor*). Dans ce cas, la fusion des éléments *infraredDetector* peut engendrer des problèmes à cause de l'incompatibilité de leurs types.

Les caractéristiques dans chaque modèle de caractéristiques sont reliées par des contraintes permettant ainsi de déterminer l'ensemble des systèmes valides. Dans l'exemple de la figure 2.9, les caractéristiques *InfraredDetector* et *volumetricDetector* sont liées par une contrainte de type ***or*** comme le montre le modèle de la figue 2.9(a), alors que les mêmes caractéristiques sont liées par une contrainte de type ***xor*** dans le modèle de la figure 2.9.(b). Pour décider de la contrainte résultante de la fusion des deux contraintes précédentes, les auteurs proposent un ensemble de règles de fusion. Ces règles calculent à partir des contraintes des modèles fusionnés les nouvelles contraintes reliant les caractéristiques du modèle de fusion. Il s'agit de calculer la contrainte prédominante des deux contraintes à fusionner selon la sémantique de fusion choisie (c'est-à-dire union ou intersection). Par exemple dans la figure 2.9.(a) les contraintes à fusionner sont de types respectifs ***or*** et ***xor***. La règle de fusion dans ce cas identifie la contrainte résultante de type ***or*** comme le montre le modèle résultant de la figure 2.9.(c).

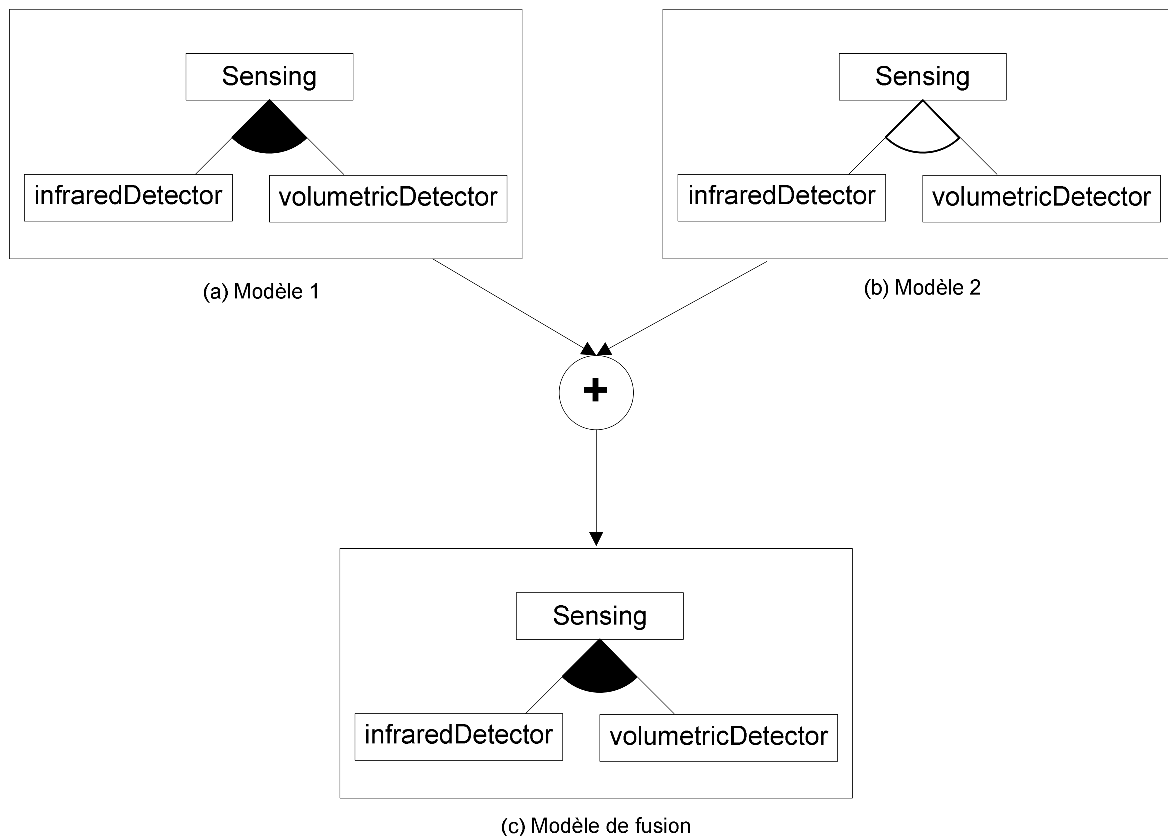


Figure 2.9 : Exemple de fusion de contraintes

Bien que les règles proposées permettent de fusionner les contraintes, elles restent limitées aux contraintes de types *and*, *or* et *xor*. Les contraintes transversales qui existent entre les caractéristiques ne sont pas traitées. Par exemple les contraintes de type *implication* ne sont pas gérées durant la fusion. Ce type de contraintes spécifie que la présence d'une caractéristique dans un système valide implique la présence d'une ou plusieurs autres caractéristiques dans le même système valide. De même pour les contraintes de type *équivalence* qui ne sont pas traitées durant la fusion. Ce type de contraintes spécifie la coprésence de toutes les caractéristiques dans un même système valide, si l'une d'entre elles est présente dans ce système et inversement. Contrairement à la fusion des contraintes, la fusion des caractéristiques qui appartiennent aux modèles d'entrée n'est pas réalisée de manière explicite. Elle est incluse dans le traitement de fusion des contraintes sans précisions supplémentaires.

- ***Opérateur d'agrégation :*** cet opérateur est utilisé pour produire de nouveaux modèles de caractéristiques en reliant d'autres modèles existants par des contraintes transversales. La proposition d'agrégation se limite à rassembler les arbres de caractéristiques en entrée sous une nouvelle caractéristique racine du modèle d'agrégation résultant. Dans l'exemple de la

figure 2.10, l'agrégation des modèles de caractéristiques *InHomeSecurity* et *Alarm* se résume simplement à créer une nouvelle racine source, *Security*, qui regroupe les deux modèles d'entrée en regroupant leurs caractéristiques racines.

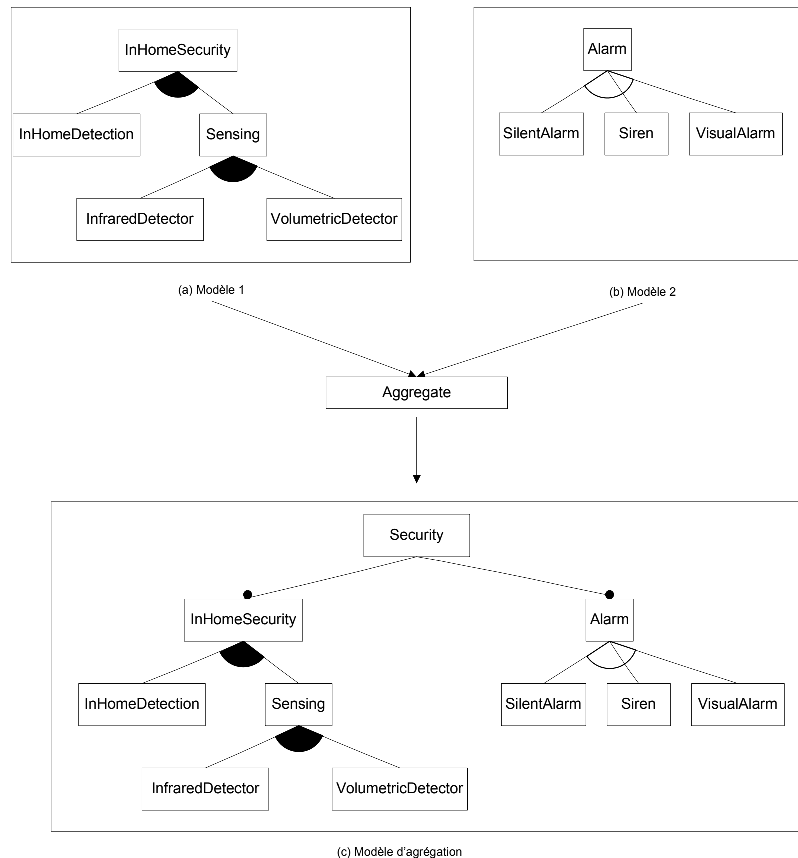


Figure 2.10 : Exemple d'agrégation de modèles de caractéristiques

Dans le contexte des modèles de caractéristiques, on ne soulève pas de questions sur la faisabilité de connecter des caractéristiques variables appartenant aux modèles dont ils réalisent l'agrégation. De même pour la gestion de la variabilité de ces caractéristiques ainsi que les contraintes qui permettent de les relier. Toutefois, ces questions représentent des points d'interrogation dans le cas d'utilisation de l'opérateur d'agrégation pour d'autres modèles comme par exemple des modèles UML. La compatibilité entre les éléments des modèles d'entrée, la gestion de la variabilité des éléments ainsi que les contraintes sont parmi les principaux problèmes à résoudre.

Approche selon la superposition de codes

Apel et al. [44][45] proposent d'organiser les éléments structurels des caractéristiques en utilisant le mécanisme de FST (Feature Structure Tree). La fusion des caractéristiques revient alors à fusionner leurs FSTs correspondantes. La proposition de la fusion se base sur le

mécanisme de superposition. La superposition permet la fusion des fragments de codes correspondants aux différentes caractéristiques. C'est un processus de composition récursive d'arbres par composition des nœuds qui se situent au même niveau en partant de la racine.
Les figures 2.11 et 2.12 montrent les modèles de caractéristiques et les fragments de code qui leurs sont correspondent respectivement. La figure 2.11 représente l'arbre de caractéristiques de la ligne de produits *InHomeSecurity* du fragment de code à gauche dans la figure. La ligne de produits inclut une détection intérieure représentée par la caractéristique *InHomeDetection* et la caractéristique de détection de mouvement *Sensing* qui inclut un détecteur infrarouge *InfraredDetector* et un détecteur volumétrique *VolumetricDetector*. Les différentes caractéristiques représentent des éléments du code correspondant.

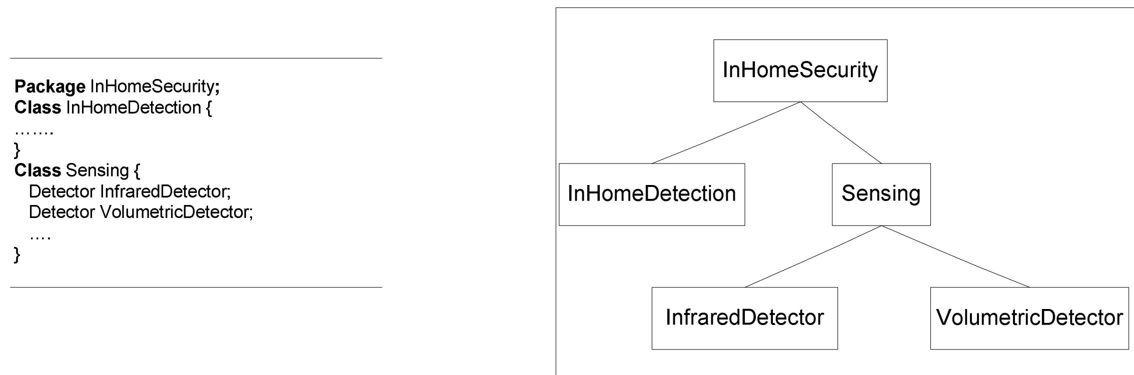


Figure 2.11: Code et modèle de caractéristiques de InHomeSecurity 1

De même, la figure 2.12 représente l'arbre de caractéristiques de la ligne de produits *InHomeSecurity* du bout de code à gauche dans la figure. La ligne de produits inclut une détection intérieure représentée par la caractéristique *InHomeDetection* et de la caractéristique de détection de mouvement *Sensing* qui inclut un détecteur infrarouge *InfraredDetector* et un détecteur de 160 degré *160DegreeDetector*. Les différentes caractéristiques représentent des éléments du code correspondant.

```
Package InHomeSecurity;
Class InHomeDetection {
…….
}
Class Sensing {
   Detector InfraredDetector;
   Detector 160DegreeDetector;
   ….
}
```

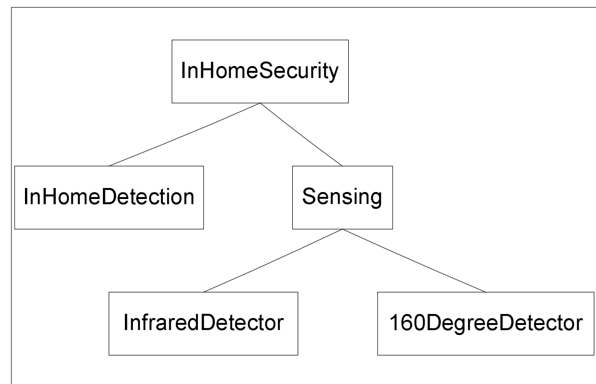


Figure 2.12: Code et modèle de caractéristiques de InHomeSecurity 2

En s'appuyant sur le mécanisme de superposition, la composition des fragments de code de la figure 2.11 et la figure 2.12 implique la composition par superposition des modèles de caractéristiques correspondants. La figure 2.13 montre le résultat de cette composition au niveau code et au niveau modèle de caractéristiques.

```
Package InHomeSecurity;
Class InHomeDetection {
…….
}
Class Sensing {
   Detector InfraredDetector;
   Detector VolumetricDetector;
   Detector 160DegreeDetector;
   ….
}
```

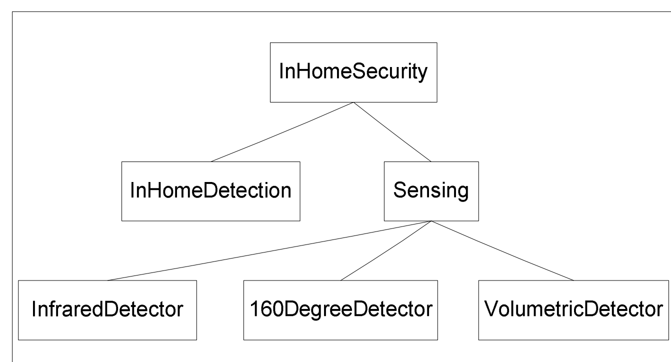


Figure 2.13:Code et modèle de caractéristiques de fusion

Un modèle de caractéristiques est « une hiérarchie de caractéristiques avec variabilité » [46]. Il peut donc être représenté par un FST avec variabilité. Cependant, la proposition ne prend pas en considération la variabilité des caractéristiques qui appartiennent aux modèles à fusionner. L'information de variabilité n'est donc pas gérée durant la fusion. De même, la proposition ne traite pas la fusion des contraintes associées aux modèles d'entrée.

Les auteurs proposent de fusionner des modèles UML non annotés par la variabilité en utilisant le mécanisme de superposition. Leur critère de correspondance entre les éléments à fusionner se base sur le nom, le type ainsi que la position dans le modèle. Cependant, ce critère représente une faiblesse dans certains cas. Prenons par exemple le modèle de structure

composite où des classes composites incluent des propriétés de nature différentes comme les ports, parts et connecteurs par exemple. Nous reprenons l'exemple de *Sensing* présenté dans la figure 2.7. Supposons que les modèles à fusionner consistent en deux classes composites nommées *Sensing* dont chacune contient deux propriétés *InfraredDetector* et *volumetricDetector*. Le premier modèle représente la propriété *volumetricDetector* par une *part* typée par une classe *VolumetricDetector*. Le deuxième modèle représente la même propriété *volumetricDetector* par un *port* typé par une classe *VolumetricDetector*. En se basant sur le critère de correspondance mentionné, c'est-à-dire le nom, le type et la position des éléments, les propriétés *volumetricDetector* des deux modèles correspondent bien qu'elles soient représentées différemment (*part* et *port*). Ceci implique des problèmes dans leur fusion.

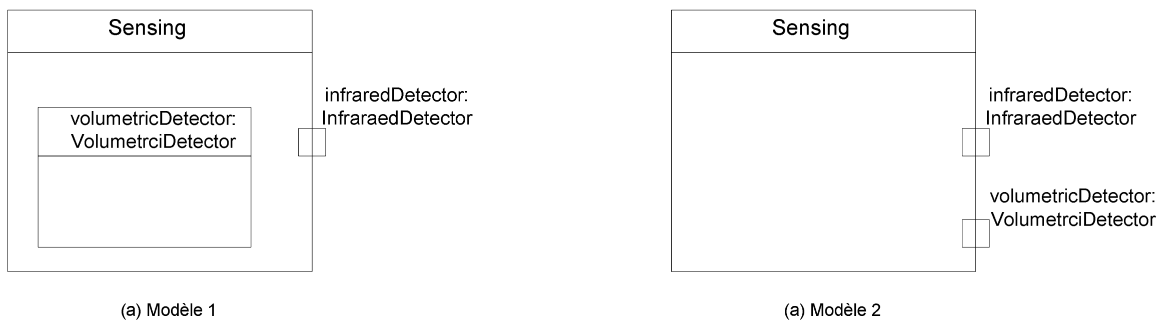


Figure 2.14: Exemple de fusion de *port* et *part*

2.2.1.3. Synthèse

La représentation des lignes de produits logiciels par le biais des modèles de caractéristiques est une pratique très répandue. Elle a l'avantage de représenter la ligne de produits logiciels de manière concise. Plusieurs travaux se sont investis dans la composition de ces modèles pour obtenir de nouvelles lignes de produits logiciels. Ces travaux offrent des mécanismes bien spécifiques à la composition des modèles de caractéristiques.

Cependant, le problème des ces approches réside en premier dans l'écart qui existe entre le niveau d'abstraction des modèles de caractéristiques et celui des modèles moins abstraits comme les modèles UML par exemple [11]. Cet écart a un impact sur les approches de composition des modèles de caractéristiques. Bien que certains critères restent valables pour ces modèles, ils se révèlent insuffisants pour des modèles plus détaillés et moins abstraits tels que les modèles UML. Citons par exemple le critère de comparaison des éléments à fusionner qui s'appuie sur les noms des caractéristiques. Il est vrai que ce critère est suffisant pour un niveau d'abstraction élevé qui est celui des modèles de caractéristiques, mais il est insuffisant

pour une représentation moins abstraite par le biais d'éléments structurels appartenant à un modèle UML. D'autre part vu que les approches de composition analysées dans cette section se focalisent sur les caractéristiques, donc elles ne traitent ni la composition des éléments structurels ni la consistance structurelle des modèles. L'information de variabilité des caractéristiques n'est pas gérée pendant la fusion par toutes les approches. De même pour les contraintes de variabilité qui ne sont pas gérées ou quand elles le sont, la majorité des approches omettent les contraintes transversales et restent limitées à certains types de contraintes.

2.2.2. Approches de modélisation orientée aspects

2.2.2.1. Modélisation

Les approches de modélisation orientée aspects séparent les éléments de modèles qui appartiennent à différentes préoccupations. Un modèle primaire ou aussi de base représente le cœur fonctionnel du système. Plusieurs modèles d'aspects sont aussi modélisés pour représenter les préoccupations transverses du système, comme par exemple la sécurité et la persistance. Les modèles d'aspects doivent alors être composés avec le modèle primaire afin d'obtenir un modèle de conception intègre [47].

La communauté de développement de logiciels par aspects porte un intérêt particulier à l'exploitation des mécanismes utilisés pour le développement des lignes de produits logiciels.

Plusieurs travaux se sont focalisés sur la capture des points communs et des points variables dans une ligne de produits logiciels en utilisant les aspects à une étape précoce [48][49][50][51] ainsi qu'à l'utilisation de la technologie de l'aspect au niveau implémentation pour représenter les caractéristiques des lignes de produits. Par exemple, Groher et Voelter [52][53][54] représentent les caractéristiques d'une ligne de produits logiciels au niveau modèle par des aspects (qui sont aussi des modèles). Au niveau implémentation, les aspects encapsulent les implémentations des caractéristiques. Pour obtenir un produit de la ligne de produits, les aspects sont composés avec la base selon une sélection des caractéristiques pour une configuration donnée. C'est ainsi qu'ils expriment la variabilité par les aspects.

2.2.2.2. Composition

Parmi les approches existantes pour la composition des modèles de lignes de produits logiciels par utilisation des technologies de l'aspect, deux approches sont sélectionnées et analysées dans cette partie.

Dans [55][56][57] les auteurs proposent de fusionner les modèles en deux phases. La première phase consiste à comparer les modèles à fusionner pour identifier les éléments qui représentent un même concept. Ces éléments sont alors dits des éléments correspondants. Ils sont ensuite fusionnés durant la phase de fusion. Pour identifier les éléments correspondants, l'approche propose de se baser sur la comparaison des signatures des éléments. Tout type d'élément possède un type de signature défini par ses propriétés syntaxiques pour assurer l'unicité des éléments. Deux éléments sont alors fusionnés s'ils ont des signatures équivalentes. Par exemple, le type de signature d'une classe UML est défini par la propriété *name* qui indique le nom de la classe ainsi que la propriété *isAbstract* qui indique si la classe est abstraite ou non. La signature de la classe *Sensing* de la figure 15 est donc définie par *{name= Sensing, isAbstract=false}*. Si deux vues de la classe *Sensing* appartiennent aux modèles à fusionner et qu'elles ont le même nom et la même valeur de la propriété *isAbstract*, elles sont alors correspondantes et elles sont fusionnées durant la phase de fusion pour former une seule classe dans le modèle de fusion résultant.

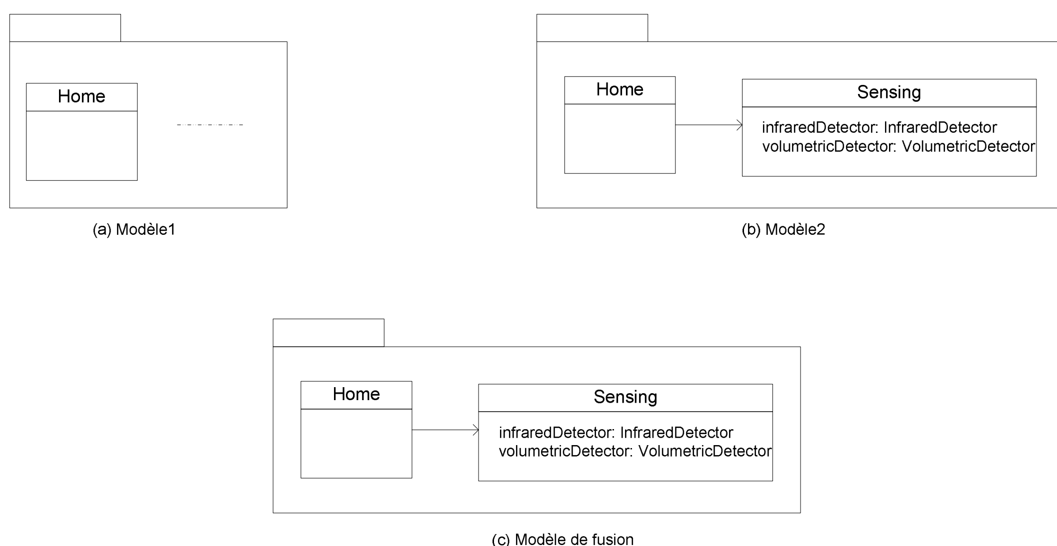


Figure 2.15: Exemple de fusion basée sur les aspects

Dans ce travail, les auteurs mènent un raisonnement local sur les modèles durant la fusion. La variabilité n'est pas traitée à ce niveau, ce qui explique l'absence de gestion de l'information de variabilité associée aux éléments des modèles à fusionner. De même pour les contraintes de variabilité qui ne sont pas traitées durant le processus de fusion proposé.

L'approche de Clarke [58][59] représente beaucoup de similitudes avec l'approche précédente. Dans cette proposition, un modèle appelé *theme* est créé pour chaque exigence du système. Ces *themes*, équivalents aux modèles d'aspects et primaires, représentent différentes vues du modèle global. Le modèle intègre est obtenu par composition des ces *themes*. Dans l'approche *Theme*, des relations de composition spécifient comment les modèles doivent être composés par identification des éléments correspondants ainsi que comment ils sont fusionnés. Le mécanisme de fusion est utilisé en se basant sur la comparaison des noms des éléments à fusionner. La description de l'utilisation de ce critère de comparaison manque de précision. Elle reste dépendante de l'utilisateur de l'approche. Cependant, nous rappelons que la comparaison des éléments à fusionner en se basant sur leurs noms demeure un critère faible et permissif comme nous l'avons illustré pour l'approche d'Acher dans la figure 2.8. De plus comme pour l'approche précédente, les auteurs raisonnent localement sur les modèles et ne traitent pas la variabilité. La fusion des modèles ne traite donc ni l'information de variabilité des éléments à fusionner ni les contraintes de variabilité qui leurs sont associées. De même que l'approche précédente, le raisonnement sur les modèles durant la fusion reste local. La variabilité n'est pas traitée à ce niveau, ce qui explique l'absence de gestion de l'information de variabilité associée aux éléments des modèles à fusionner. De même pour les contraintes de variabilité qui ne sont pas traitées durant le processus de fusion proposé.

2.2.2.3. Synthèse

L'utilisation des techniques de développement par aspect des logiciels dans le contexte des lignes de produits logiciels prend de plus en plus d'ampleur. En effet, les différentes exigences de la ligne de produits sont modélisées sous forme d'aspects (qui sont des modèles) et d'un modèle de base. Pour obtenir un produit de la ligne de produits logiciels, le modèle de base est composé avec les aspects qui représentent les exigences du produit souhaité. Les mécanismes de composition à ce niveau se basent sur l'aspect structurel des modèles composés. Cependant le raisonnement de ces approches au niveau composition reste local et ne traite donc ni l'information de variabilité des éléments structurels ni les contraintes de variabilité. De même la sémantique de composition n'est pas évoquée par ces approches.

2.2.3. Approches dirigées par les modèles annotatrices de variabilité

2.2.3.1. Modélisation

Certaines approches de développement de lignes de produits logiciels utilisent des annotations pour identifier les éléments variables dans le modèle. Nous distinguons les approches qui représentent les lignes de produits logiciels par le biais de modèles UML [11]. Plusieurs approches dirigées par les modèles utilisent des stéréotypes UML pour annoter les éléments variables qui appartiennent aux modèles de lignes de produits logiciels [3][21][22][23][4].

Par exemple, Clauss [21][24] propose un profil UML pour permettre l'expression de la variabilité dans les modèles UML qui représentent des lignes de produits logiciels. Ce profil indique que tout élément du modèle peut faire l'objet d'un point de variation. Un point de variation localise la variabilité dans un modèle et ses différentes réalisations décrivent plusieurs variantes. Le point de variation contient plusieurs variantes qui déterminent les caractéristiques de la variabilité. Une variante est toujours attachée à un point de variation. Le point de variation implique plusieurs valeurs étiquetées pour déterminer le moment de la réalisation du point de variation et la cardinalité des variances ainsi que le nombre de variances qui lui correspondent. Le stéréotype «*optional*» indique qu'un élément peut être supprimé dans le modèle dérivé.

Dans le travail de Ziadi [22] en plus de l'utilisation du modèle de caractéristiques, cette approche utilise des extensions d'UML pour modéliser la variabilité au niveau structurel grâce à deux mécanismes à savoir l'optionalité et la variation ainsi qu'au niveau comportemental moyennant trois mécanismes qui sont l'optionalité, la variation et la virtualité. Au niveau structurel (aspect statique) par exemple deux stéréotypes sont proposés:

- L'optionalité: par l'utilisation du stéréotype «*optional*» ce qui signifie que certaines propriétés sont optionnelles dans certains produits et donc elles peuvent être supprimées. L'optionalité n'est pas modélisée au niveau des associations entre les classes car une classe optionnelle signifie que toutes les associations auxquelles elle participe sont aussi optionnelles. L'optionalité d'une classe s'étend sur tous ses attributs et ses opérations, idem pour un paquetage optionnel dont l'optionalité s'étend sur tous ses éléments.
- La variation: par l'introduction des stéréotypes «*variation*» et «*variant*» associés respectivement aux classes abstraites et aux sous-classes concrètes.

Au niveau comportemental (aspect dynamique): diagrammes de séquence

- L'optionalité: par l'introduction du stéréotype «*optionalLifeline*» qui s'applique sur les messages reçus et envoyés des objets optionnels dans le diagramme de séquence. Le stéréotype «*optionalInteraction*» spécifie l'optionalité des interactions.
- La variation: par l'introduction du stéréotype «*variation*» appliqué sur une interaction qui référence un ensemble de sous-interactions stéréotypées «*variant*».
- La virtualité: par l'introduction du stéréotype «*virtual*» qui une fois appliqué sur une interaction, signifie que le comportement de celle-ci peut être redéfini par une autre interaction de raffinement associée à un produit particulier. Le comportement de l'interaction virtuelle sera remplacé pendant la dérivation de produit par le comportement de l'interaction de raffinement associée au produit.

Kobra [3] est basée sur les composants. On ne parle plus de modèle de caractéristiques mais plutôt d'arbre de composants dont le composant racine représente le système et les sous composants représentent les éléments constitutifs du système. Elle utilise un seul stéréotype «*variable*» pour marquer les éléments variables de type paquetage, classe, attribut, opération, transition et interaction. Au niveau statique, l'approche traite les diagrammes de classes, au niveau comportemental (dynamique), les diagrammes de séquences et machine à états. L'approche introduit un modèle de décisions qui remplace le rôle du diagramme de caractéristiques et qui est construit à partir du modèle de famille de systèmes et des spécifications de l'ensemble des systèmes visés. Pour chaque décision le concepteur doit choisir une résolution qui affecte alors le modèle de famille de systèmes sous-jacent et peut renvoyer l'utilisateur vers de nouvelles décisions.

L’approche SEQUOIA (précédemment nommée SyF) [25] utilise le diagramme de caractéristiques pour la capture des exigences. Elle offre le moyen d'exprimer la variabilité au niveau du modèle UML de la ligne de produits logiciels grâce au stéréotype «*VariableElement*» (annexe A). Les contraintes de variabilité sont exprimées pour identifier les produits logiciels valides qui peuvent être obtenus à partir du modèle de la ligne de produits logiciels. Nous citons à titre d’exemple une contrainte d’implication entre deux éléments variables (annexe A):

Implication (elem1, elem2) : une contrainte de type *Implication* spécifie que la présence de l’élément *elem1* dans un produit valide implique la présence de l’élément *elem2* dans le même produit valide de la ligne de produits et inversement.

2.2.3.2. Composition

UML propose un mécanisme de fusion de modèles qui pourrait être envisagé pour traiter la composition de modèles annotés. Le mécanisme de fusion de paquetage d'UML est utilisé pour combiner des éléments qui représentent le même concept et ayant le même nom et le même méta-type. Cependant, le mécanisme de fusion d'UML ne considère pas l'aspect variabilité des éléments à fusionner. De ce fait, ni l'information de variabilité des éléments ni les contraintes de variabilité ne sont gérées par ce mécanisme d'UML. Celui-ci est focalisé sur la fusion des éléments des modèles UML dont les éléments structurels auxquels nous nous intéressons. En effet, la spécification d'UML donne des définitions détaillées sur les critères de fusion selon la syntaxe des éléments à fusionner. Notamment la spécification UML dans ce cas définit les règles de fusion des éléments *Property* de manière générale. Ces définitions sont nombreuses mais ne sont pas assez précises dans la pratique pour certains éléments comme par exemple la différentiation entre *port* et *part*.

Le mécanisme de fusion d'UML est un mécanisme intéressant pour les approches basées sur les modèles UML. Cependant, ce mécanisme ne traite pas la variabilité donc ni l'information de variabilité des éléments, ni les contraintes de variabilité ne sont gérées durant la fusion. De plus, les critères de fusion spécifiés pour les éléments structurels nécessitent plus de précision pour éviter d'éventuelles pertes d'information ou l'obtention de modèles défaillants à l'issue de la fusion.

Les approches dirigées par les modèles et annotatrices de variabilité représentent une ligne de produits logiciels dans sa globalité par le biais d'un seul modèle UML. Elles utilisent des stéréotypes bien spécifiques pour distinguer les éléments variables dans ce modèle. L'obtention des modèles de produits est alors réalisée en effectuant des sélections sur les éléments annotés comme variables. Un modèle de produit est donc constitué des éléments communs et d'une sélection des éléments variables. Ces approches ont l'avantage de fournir plusieurs mécanismes qui permettent de distinguer explicitement les éléments variables dans un modèle de lignes de produits logiciels. Ceci facilite aux concepteurs le développement des lignes de produits logiciels et incite donc à les utiliser.

Bien qu'elles soient nombreuses à développer les lignes de produits logiciels en s'appuyant sur des modèles UML qui distinguent les éléments variables par des annotations (les stéréotypes) précédemment présentées, le problème avec ce type d'approches est qu'elles ne fournissent pas des mécanismes explicites et spécifiques pour la composition des modèles UML annotés par des stéréotypes de variabilité. Des efforts doivent être faits pour permettre

la réutilisation de ces modèles dans la composition afin d’obtenir de nouveaux modèles de lignes de produits logiciels.

2.2.4. Bilan et positionnement

Dans cette section, nous avons présenté une sélection de travaux dans la composition des lignes de produits logiciels. Nous avons classé ces approches selon les modèles qu’elles utilisent pour représenter les lignes de produits logiciels.

La catégorie d’approches de modèles de caractéristiques. Ces approches modélisent les exigences des lignes de produits logiciels par une hiérarchie de caractéristiques. Plusieurs travaux ont proposé des mécanismes de composition des modèles de caractéristiques [38][39][60][44]. Nous avons montré que ces catégories ne s’intéressent ni à l’aspect structurel des lignes de produits ni à sa consolidation. Le calcul de l’information de variabilité des éléments résultants de la composition n’est pas toujours réalisée comme par exemple dans le travail de Apel [44]. Elle est réalisée implicitement dans le travail d’Acher [42]. De même, la fusion des contraintes de variabilité ne sont pas traitées par toutes les approches durant la composition [44], en particulier les contraintes transversales [42]. La sémantique de composition n’est pas explicite dans tous les travaux proposes.

La catégorie d’approches qui gèrent les lignes de produits logiciels par les mécanismes de développement orienté aspects. Les mécanismes de composition proposés s’intéressent particulièrement à l’aspect structurel des éléments à composer [55]. Cependant, l’information de variabilité n’est pas gérée durant la composition comme est le cas pour les contraintes de variabilité.

La catégorie annotatrices de variabilité [22][3][25]. Ces approches proposent des stéréotypes UML pour annoter les éléments variables d’un modèle UML de ligne de produits logiciels. Cependant, cette catégorie d’approche ne fournit pas de mécanismes pour la composition des modèles de lignes de produits logiciels. Des travaux de recherche doivent être effectués pour la composition.

Notre contribution s’inscrit parmi les approches de composition des modèles de lignes de produits logiciels annotatrices de variabilité. Nous utilisons la méthodologie développée dans le laboratoire pour l’expression et la gestion de la variabilité (SEQUOIA), précédemment présentée, pour le développement de lignes de produits logiciels [25] vu qu’elle fournit des mécanismes spécifiques pour distinguer les éléments structurels variables. Nous proposons de représenter le modèle de ligne de produits logiciels par un modèle UML où les éléments variables sont annotés par le stéréotype «*VariableElement*» (voir annexe A). Nous allons

montrer dans la suite l'importance de la consolidation des modèles structurels des lignes de produits logiciels et comment l'assurer. Nous allons aussi montrer comment calculer l'information de variabilité des éléments résultants de la composition. Vu que les contraintes de variabilité jouent un rôle principal dans la définition de l'ensemble de produits valides d'une ligne de produits logiciels, nous allons montrer comment ces contraintes sont gérées pour satisfaire la sémantique de composition.

2.3. Conclusion

Le but de ce chapitre est d'analyser les travaux qui s'intéressent à la composition des modèles de lignes de produits logiciels. Pour ce faire, la première section a été consacrée à la présentation de l'ingénierie des lignes de produits logiciels. Nous avons exposé ses principaux apports et activités. La deuxième section a été consacrée à l'analyse des approches de composition des lignes de produits logiciels. Dans cette section, nous avons classé les approches de composition selon les modèles qu'elles utilisent dans la représentation des lignes de produits logiciels. Pour chaque catégories, nous avons présenté les modèles utilisés, puis nous avons discuté les mécanismes de composition proposés selon un ensemble de critères à savoir la gestion de l'aspect structurel et de la consistance des modèles, l'information de variabilité des éléments, les contraintes de variabilité et la sémantique de composition visée. Nous avons exprimé notre positionnement par rapport aux approches présentées en se référant à ces critères. En effet, les approches basées sur les modèles de caractéristiques offrent plusieurs mécanismes pour la composition mais leurs solutions restent dépendantes de la nature des modèles de caractéristiques et permissifs pour être utilisés tels quels sur d'autres modèles comme les modèles UML. Les approches basées sur la modélisation orientée aspects s'intéressent à la fusion des éléments structurels mais leur vision reste locale. Ni l'information de variabilité des éléments structurels ni les contraintes de variabilité ne sont considérées durant la fusion. Les approches dirigées par les modèles et annotatrices de variabilité offrent des mécanismes intéressants pour le développement des lignes de produits logiciels. Cependant, elles ne fournissent pas de mécanismes spécifiques pour la composition. Ces approches se basent sur les modèles UML, d'où l'importance de considérer le mécanisme de fusion proposé par la norme UML. Ce mécanisme offre la possibilité de fusionner des modèles UML mais nécessite plus de précisions et ne traite pas l'information de variabilité des éléments à fusionner. De même pour les contraintes de variabilité qui ne sont pas gérées durant la fusion. Notre contribution s'intéresse à la composition des modèles de lignes de produits logiciels incluant des annotations de

variabilité. Dans le chapitre suivant, nous allons présenter nos choix de modélisation et souligner l'importance de la consolidation de la structure des modèles manipulés. Nous présenterons ensuite un aperçu global sur les contributions de composition proposées dans ce manuscrit.

CHAPITRE 3

Choix de modélisation et aperçu global des contributions

3.1. Modèle de ligne de produits logiciels : vue de structure composite 39
3.2. Consolidation de modèles de lignes de produits logiciels.. 41
3. 2.1. Règle de connexion à un port simple (Reg1) 44
3.2.2. Règle de connexion à un port multiple (Reg2)... 45
3.2.3. Règle de port variable attaché sur une part variable (Reg3)...................................... 47
3.2.4. Règle de port variable connecté à un port attaché sur une part variable (Reg4) 48
3.2.5. Règle de connecteur variable reliant deux ports simples (Reg5) 50
3.2.6. Règle de connecteur variable reliant deux ports dont au moins un est multiple (Reg6) 51
3.2.7. Synthèse.. 55
3.3. Formes de composition 55
3.4. Conclusion……………………………………………………………………….58

Dans ce chapitre, nous présentons le choix retenu pour la modélisation des lignes de produits logiciels. Nous proposons deux formes de composition de modèles de lignes de produits et nous donnons un aperçu global sur ces mécanismes.
Ce chapitre est divisé en trois sections. La première section explicite les choix effectués pour la modélisation des lignes de produits logiciels et présente les structures composites d'UML qui seront l'objet de composition dans le reste du manuscrit. La deuxième section présente un ensemble de règles pour la consolidation des modèles de lignes de produits logiciels. La troisième section propose un aperçu global des deux formes de composition proposées dans ce travail.

3.1. Modèle de ligne de produits logiciels : vue de structure composite

Les diagrammes de structures composites d'UML 2 permettent de modéliser et d'identifier les parties réutilisables d'un modèle de conception donné. Pour notre travail, nous avons choisi de modéliser une ligne de produits logiciels par un diagramme de structure composite d'UML dans l'optique de renforcer la réutilisation des parties de modèles. Une ligne de produits logiciels est donc représentée par une classe composite ayant des éléments externes représentés par des *ports* et encapsulant un ensemble de *parts*. Toute *part* possède un ensemble de *ports* qui matérialisent ses points de connexion. Des connecteurs, dits d'assemblage, permettent de relier les points de connexion entre les différentes *parts*. D'autres connecteurs, dits de délégation, permettent à un port interne de déléguer à un autre *port* externe. Un *port* est dit simple s'il a une seule connexion. Ceci veut dire qu'il est connecté à un seul *port* via un seul connecteur. Un *port* est dit multiple s'il a plus d'une seule connexion, c'est-à-dire qu'il est connecté à un ou plusieurs ports via plus d'un seul connecteur. La modélisation d'une ligne de produits logiciels sous une vue de structure composite suppose que les options soient modélisées par les concepts que nous venons de présenter. Ceci implique que la variabilité pourrait être appliquée à tous les concepts présentés pour la structure composite d'une ligne de produits. D'autre part, la présence des éléments variables dans un modèle de produit donné doit respecter un ensemble de contraintes. Cet ensemble de contraintes représente les différents liens de dépendance entre les éléments variables dans un modèle de ligne de produits. Il définit l'ensemble de ses produits (ou systèmes) valides obtenus à partir du modèle de ligne de produits.

Exemple : Modèle de ligne de produits de sécurité «Détecteur de présence avec alarme»

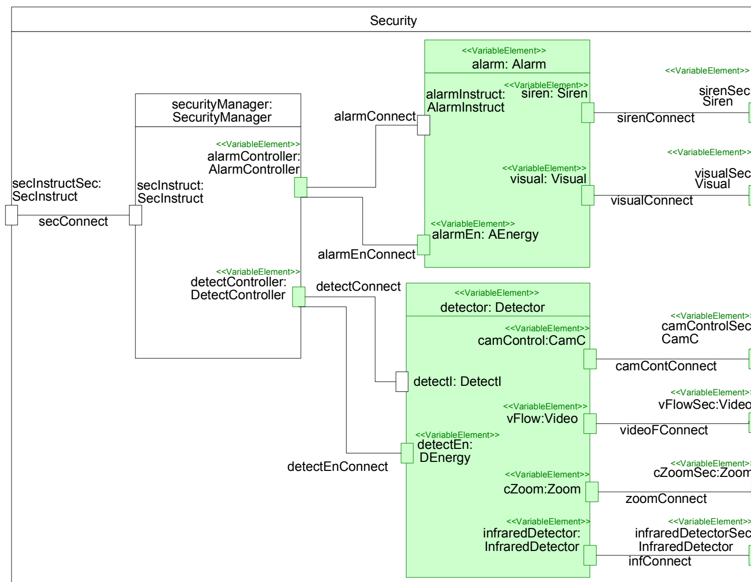


Figure 3.1 : Diagramme de structure composite de la ligne de produits Security

L'exemple que nous prenons est celui d'une ligne de produits qui détecte toute présence suspecte de deux manières différentes: via une caméra surveillance qui effectue un enregistrement vidéo ou via un détecteur de mouvements infrarouge. Dès qu'une présence suspecte est détectée, une alarme est déclenchée. Deux types d'alarmes sont à prendre en considération, à savoir l'alarme en mode sirène et l'alarme visuelle. Le détecteur de présence et l'alarme sont contrôlés par un gestionnaire de sécurité. Ce contrôle est optionnel car il dépend de la présence du détecteur et de l'alarme qui sont eux-mêmes optionnels. La figure 3.1 représente un modèle de la structure composite de la ligne de produits *Security*. Ce modèle indique que la ligne de produits *Security* propose l'option de détection et l'option d'alarme modélisées par les parts respectives *detector* et *alarm* dénotées comme variables. Ces deux parts variables sont gérées par une troisième part fixe, *securityManager*, à travers des ports optionnels ; *alarmController* pour contrôler à la fois la part *alarm* via le port fixe *alarmInstruct* ainsi que l'énergie qu'elle utilise via le port variable *alarmEn*. Le port *detectController* permet de gérer la part *detector* via le port fixe *detectI* ainsi que l'énergie utilisée via le port variable *detectEn*. La détection est réalisée via les ports variables *camControl*, *camControlSec, VFlow*, *VFlowSec, Czoom* et *CzoomSec* afin d'assurer la caméra surveillance. Pour la détection de mouvements, un capteur infrarouge est modélisé par les ports *infraredDetector* et *infraredDetectorSec*. Les options d'alarme sont réalisées par les ports *siren* et *sirenSec* pour la sirène et par les ports variables *visual* et *visualSec* pour

l'alarme visuelle. La ligne de produits S*ecurity*, modélisée dans la figure 3.1, expose ses services via le port fixe *detecInstructSec*. Les connecteurs à l'instar de *sirenConnect*, *visualConnect* et *alarmConnect,* réalisent la connexion à travers les ports.

Afin que les éléments variables soient présents dans un modèle de produit valide, ils doivent respecter un ensemble de contraintes que nous identifions comme suit :

- C_1= *Implication (alarm, detector),* C_1 est la contrainte qui relie les deux parts variables *alarm* et *detector*. *Implication* est le type de la contrainte C_1 qui spécifie que si la part *alarm* est présente dans un modèle de produit valide, alors la part *detector* doit l'être aussi.

- C_2= *AtLeastOne (siren, visual),* C_2 est la contrainte qui relie les deux ports variables *siren* et *visual*. *AtLeastOne* est le type de la contrainte C_2 et qui représente le type de contrainte *OU*. Cette contrainte indique qu'un modèle de produits doit inclure au moins l'option *siren* ou *visual* ou bien les deux.

- C_3= *AtLeastOne (camControl, infraredDetector),* C_3 est la contrainte qui relie les deux ports variables *camControl* et *infraredDetector*. *AtLeastOne* est le type de la contrainte C_3. Cette contrainte indique qu'un modèle de produits doit inclure au moins l'option *camControl* ou *infraredDetector* ou bien les deux.

- C_4= *Equivalence (camControl, vFlow, cZoom),* C_4 est la contrainte qui relie les trois ports variables *camControl*, *vFlow* et *cZoom*. *Equivalence* est le type de la contrainte C_4. Cette contrainte indique qu'un modèle de produit doit inclure obligatoirement les trois ports *camControl*, *vFlow* et *cZoom* si l'un des trois est présent et inversement.

3.2. Consolidation de modèles de lignes de produits logiciels

Sachant qu'une ligne de produits est modélisée sous une vue de structure composite, le respect des contraintes de variabilité spécifiées est nécessaire mais pas suffisant. Des contraintes supplémentaires relatives à l'aspect structurel de la ligne de produits doivent être créées et respectées en plus des contraintes déjà spécifiées afin d'assurer la réalisation des options de la ligne de produits. Si un modèle de ligne de produits respecte uniquement les contraintes de variabilité sans considérer les propriétés structurelles engendrées par la présence de la variabilité dans le modèle, les modèles de produit résultants seront structurellement incomplets et par conséquent non capables de réaliser les fonctionnalités qui leurs sont attribuées.

La ligne de produits *Security* dont le modèle est présenté dans la figure 3.1 doit respecter les contraintes C_1, C_2, C_3 et C_4 associées à ses éléments variables. La résolution de ces contraintes donne l'ensemble des produits valides. La figure 3.2 représente un modèle de produit de la ligne de produits *Security* valide du point de vue des contraintes *C1, C2, C3* et *C4*. Ce modèle indique que le produit comporte les parts *detector* et *alarm* ainsi que leurs différents ports. Par exemple la figure 3.2 montre que les options de sirène et d'alarme visuelle sont présentes via les ports respectifs *siren* et *visual*. De même pour les options de détection représentées par les ports *camControl*, *vFlow, cZoom* et *infraredDetector*. Le port *alarmEn* est présent dans le modèle de produit pour permettre la réception de l'énergie.

Le modèle de produit illustré ci-dessous est valide en terme de contraintes utilisateur. Cependant, la classe composite *Security*, telle que spécifiée dans la figure 3.2, est incapable d'assurer ses fonctionnalités d'alarme et de détection: sa structure incomplète. En effet, l'absence des ports externes, *camControlSec*, *vFlowSec, cZoomSec* et *infraredDetectorSec*, empêche la classe *Security* de communiquer avec l'environnement extérieur pour réaliser les fonctions de détection. De même pour les ports externes *sirenSec* et *visualSec* sont absents du modèle de produit ce qui empêche la classe *Security* de communiquer avec l'environnement extérieur pour réaliser les fonctions d'alarme. De plus, l'absence de ces ports externes entraine l'absence des connecteurs qui les relient à leurs ports respectifs. D'autre part l'absence du port *alarmController* via lequel la part *securityManager* doit gérer la part *alarm* engendre l'isolation de celle-ci et empêche donc de contrôler son fonctionnement ainsi que son énergie.

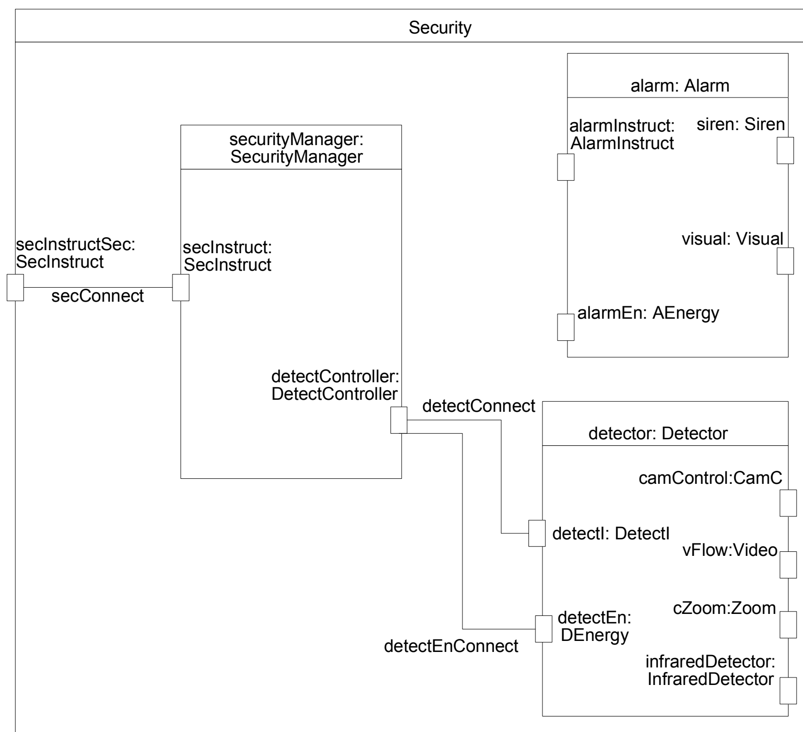


Figure 3.2 : Modèle de produit valide sans prise en compte des contraintes structurelles

Pour remédier à un tel problème, nous proposons un catalogue de règles de consolidation de modèles visant à assister le concepteur de la ligne de produits. Ces règles sont appliquées sur le modèle de ligne de produits afin d'assurer que les modèles de produit dérivés soient structurellement cohérents et complets. Ainsi, le concepteur de la ligne de produits ne sera plus amené à vérifier la variabilité induite par la structure en plus de la variabilité spécifiée par l'utilisateur.

Pour élaborer la solution de consolidation proposée dans la suite, nous avons procédé selon la méthode suivante ; nous avons commencé par identifier certaines règles de cohérence liées au méta-modèle UML spécifié pour la construction de diagrammes de structures composites [11]. Nous avons ensuite considéré des règles métier relatives à la construction de structures (architectures). A partir de ces deux ensembles de critères, nous avons déduit l'ensemble des règles de consolidation pour la construction de modèles de lignes de produits logiciels. Ces règles ne sont pas générales puisqu'elles dépendent des règles métier qui ne sont pas invariantes mais dépendent du domaine ainsi que des critères de bonnes constructions établies par le concepteur.

3. 2.1. Règle de connexion à un port simple (Reg1)

(Reg1) :

Si deux ports simples sont connectés et qu'au moins un des deux est variable, alors le deuxième port doit aussi être variable. Une contrainte de type équivalence doit être créée entre les deux ports et spécifier que si l'un des deux ports est présent dans un modèle de produit dérivé alors l'autre le sera aussi et inversement.

Exemple:

La figure 3.3 présente la classe composite *Security* modélisant une ligne de produits (figure 3.3.(a)) ainsi qu'un modèle de produit dérivé (figure 3.3.(b)). Le modèle de ligne de produits ne possède aucune contrainte structurelle.

Dans la figure 3.3.(a) le port *siren* attaché sur la part *alarm* est connecté au port *sirenSec* via un connecteur. Nous pouvons remarquer que les ports *siren* et *sirenSec* sont variables et qu'aucune contrainte structurelle n'est associée aux éléments du modèle, (voir la figure 3.3.(a)). La dérivation du modèle de ligne de produits, illustré dans 3.3.(a), montre l'obtention du modèle de produit présenté dans la figure 3.3.(b). On constate que le modèle de produit obtenu est structurellement incomplet. Ceci s'explique par l'absence de contrainte structurelle qui relie les deux ports *siren* et *sirenSec*.

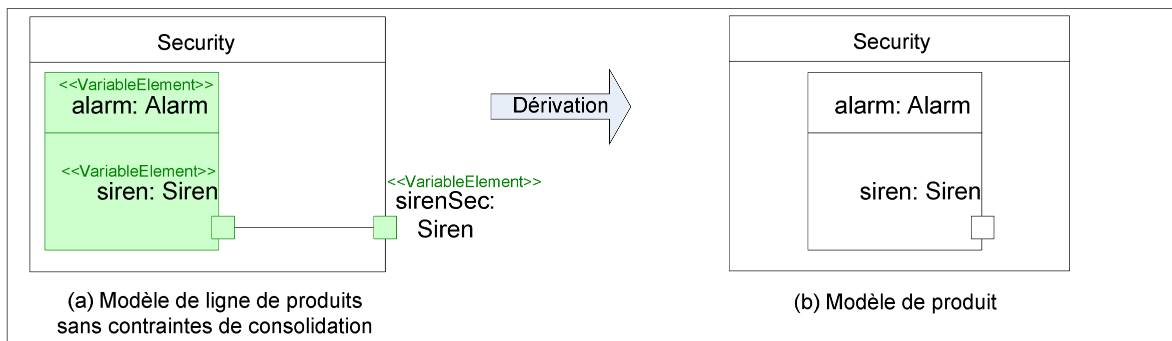


Figure 3.3 : Avant application de la règle de consolidation *(Reg1)*

Pour remédier à ce problème, nous appliquons la règle de consolidation *(Reg1)* dont nous pouvons constater les résultats sont présentés dans la figure suivante. La figure 3.4.(a) montre que l'application de *(Reg1)* entraine l' ajout de la contrainte d'équivalence associée aux ports *siren* et *sirenSec*. La contrainte C_5 impose la coprésence des deux ports, dans le cas où un des deux doit être présent dans le modèle de produit dérivé. Cette contrainte écarte la possibilité d'obtenir le modèle de produit incomplet de la figure 3.3.(b). Ainsi le modèle de produit obtenu est complet : il contient bien les deux ports *siren* et *sirenSec* grâce à la contrainte structurelle C_5.

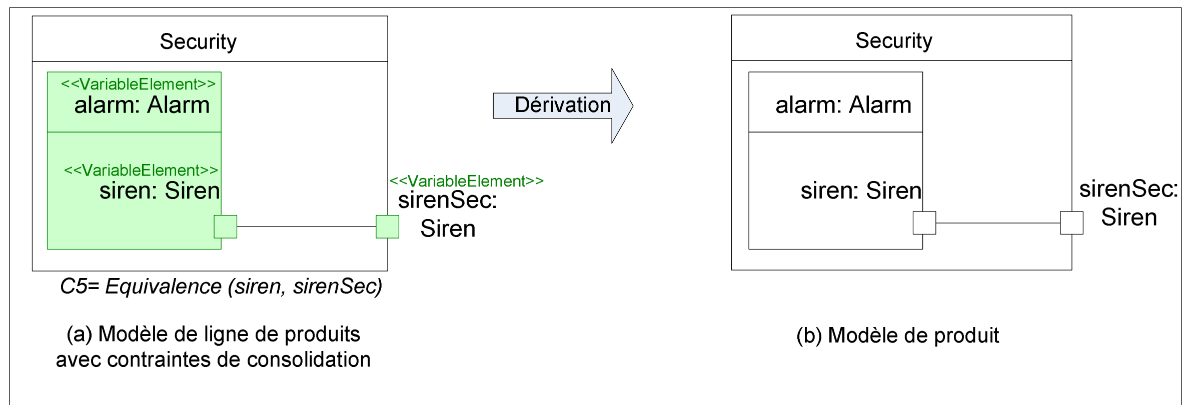


Figure 3.4 : Après application de la règle de consolidation *Reg1*

3.2.2. Règle de connexion à un port multiple (Reg2)

(Reg2) :

Si un port est connecté à un autre port multiple et qu'au moins un des deux est variable, alors le deuxième port doit aussi être variable. Une contrainte de type implication doit être créée pour spécifier que la présence du premier port dans un modèle de produit dérivé implique la présence du port multiple.

Exemple:

La figure 3.5 présente la classe composite *Security* représentant une ligne de produits qui ne possède aucune contrainte structurelle associée à ses éléments (figure 3.5.(a)) ainsi qu'un modèle de produit dérivé (figure 3.5.(b)).

Dans la figure 3.5.(a), le port *alarmController* attaché sur la part *securityManager* est un port multiple connecté aux deux ports *alarmInstruct* et *alarmEn* qui sont de leur part attachés à la part *alarm*. Nous pouvons remarquer que les ports *alarmEn* et *alarmController* sont variables et qu'aucune contrainte structurelle n'est associé aux éléments du modèle, (voir la figure 3.5.(a)). La dérivation du modèle de ligne de produits, illustré dans 3.5.(a), montre l'obtention du modèle de produit présenté dans la figure 3.5.(b). On constate que le modèle de produit obtenu est structurellement incomplet. En effet, les ports *alarmInstruct* et *alarmEn* sont présents afin d'exposer les services de la part *alarm*. Cependant, le port *alarmController* qui transmet les instructions à la part *alarm* n'est pas présent dans le modèle de produit. Ceci s'explique par l'absence de contrainte structurelle qui relie les deux ports *alarmInstruct* et *alarmEn* au port *alarmController*.

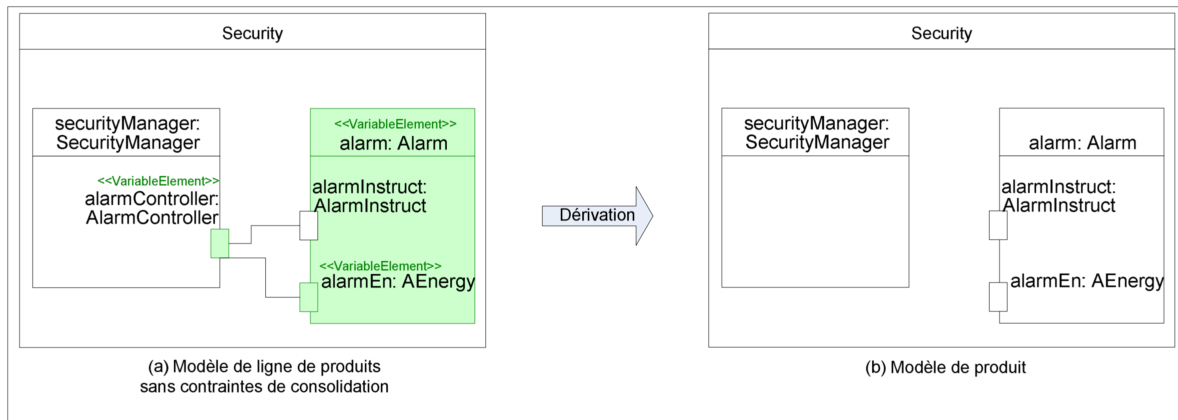


Figure 3.5 : Avant application de la règle de consolidation *(Reg2)*

L'application de la règle de consolidation (Reg2) permet d'éviter ce type de problème. La figure 3.6 montre l'application de *(Reg2)* par ajout de la contrainte d'implication associée aux ports *alarmEn* et *alarmController* ainsi que de la contrainte d'implication associée aux ports *alarmInstruct* et *alarmController* après que *alarmInstruct* est devenu variable (voir figure3.6.(a)). La contrainte C_6 impose l'ajout du port *alarmController* en cas de présence du port *alarmInstruct* dans un même modèle de produit. De même pour la contrainte C_7 qui impose l'ajout du port *alarmController* en cas de présence du port *alarmEn* dans un même modèle de produit. Ces contraintes assurent que le modèle de produit de la figure 3.5.(b) ne puisse être obtenu. La modèle de produit obtenu dans la figure 3.6.(b) est complet et contient le port *alarmController* grâce à l'ajout des contraintes structurelles C_6 et C_7.

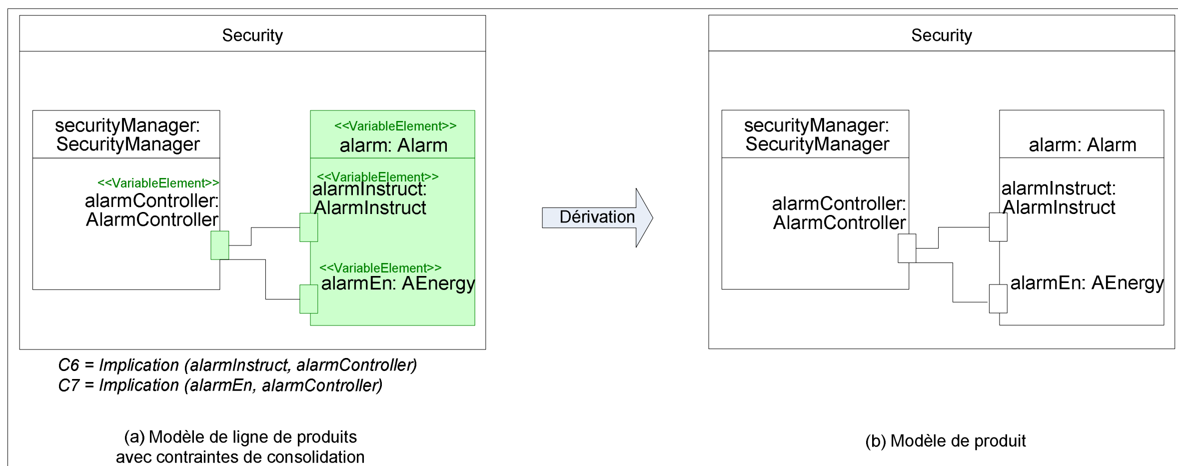


Figure 3.6 : Après application de la règle de consolidation *Reg2*

3.2.3. Règle de port variable attaché sur une part variable (Reg3)

(Reg3) :

Si un port est variable et qu'il est attaché à une part variable, une contrainte de type implication doit être créée pour spécifier que la présence du port implique la présence de la part à laquelle il est attaché.

Exemple:

La figure 3.7 présente la classe composite *Security* représentant une ligne de produits, qui ne possède aucune contrainte structurelle associée à ses éléments (figure 3.7.(a)), ainsi qu'un modèle de produit dérivé (figure 3.7.(b)).

Dans la figure 3.7.(a) le port *siren* est variable et attaché à la part variable *alarm*. On remarque l'absence de toute contrainte structurelle qui relie le port *siren* à la part d'*alarm*, (voir la figure 3.7.(a)). La dérivation du modèle de ligne de produits, illustre en 3.7.(a) l'obtention du modèle de produit présenté dans la figure 3.7.(b). La figure présente un modèle de produit dans lequel le port *siren* doit être présent contrairement à la part *alarm*. Cependant, la figure 3.7.(b) montre l'absence de *siren* et *alarm*. Ceci s'explique par le fait que le port *siren* ne peut être visualisé et sa présence ne peut avoir un sens que si la part *alarm* est elle aussi présente dans le modèle. L'absence de contrainte structurelle qui relie le port *siren* à la part *alarm* sur laquelle il est attaché peut engendrer des modèles incomplets comme dans la figure 3.7.(b).

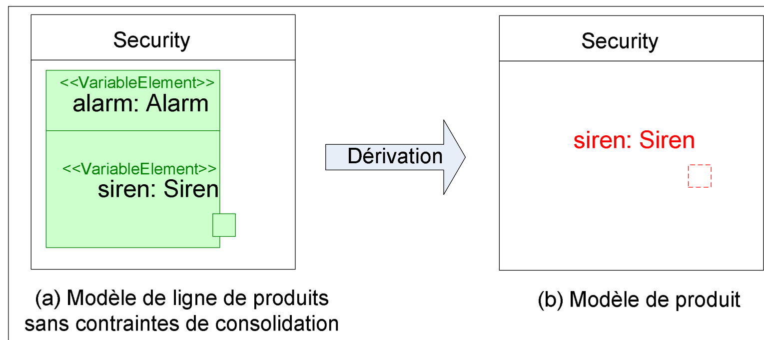


Figure 3.7 : Avant application de la règle de consolidation *Reg3*

Pour remédier à ce problème, nous appliquons la règle de consolidation *(Reg3)* dont on peut constater les résultats sur la figure suivante. La figure 3.8 montre l'application de *(Reg3)* par ajout de la contrainte d'implication associée au port *siren* et la part *alarm* à laquelle il est attaché (voir figure3.8.(a)). La contrainte C_8 impose la présence de la part *alarm* si le port *siren* est présent dans un modèle de produit. Cette contrainte assure que le modèle de produit incomplet de la figure 3.7.(b) ne puisse être obtenu. Le modèle de produit obtenu, présenté en

figure 3.8.(b), est complet et contient la part *alarm* et le port *siren* grâce à l'ajout de la contrainte structurelle C_8.

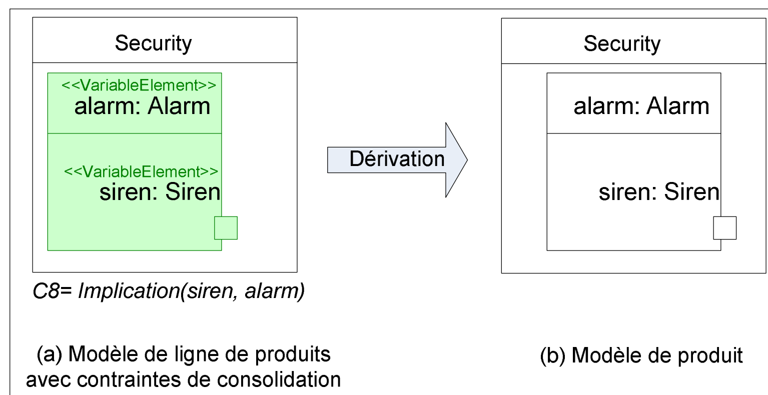


Figure 3.8 : Après application de la règle de consolidation *Reg3*

3.2.4. Règle de port variable connecté à un port attaché sur une part variable (Reg4)

(Reg4) :

Si un port variable est connecté à un port attaché sur une part variable, alors une contrainte de type implication est créée pour spécifier que la présence du premier port implique celle de la part.

Exemple:

La figure 3.9 présente la classe composite *Security* représentant une ligne de produits qui ne possède aucune contrainte structurelle associée à ses éléments (figure 3.9.(a)) ainsi qu'un modèle de produit dérivé (figure 3.9.(b)).

Dans la figure 3.9.(a) le port *alarmController* est variable et il est connecté au port *alarmInstruct*. Ce dernier, quant à lui, est attaché sur la part variable *alarm*. Aucune contrainte structurelle n'est associée aux éléments du modèle, (voir la figure 3.9.(a)). La dérivation du modèle de ligne de produits, illustré dans 3.9.(a), montre l'obtention du modèle de produit présenté dans la figure 3.9.(b). On constate que le modèle de produit obtenu est structurellement incomplet. En effet, le port *alarmController* est présent afin de transmettre les instructions à la part *alarm* alors que la part *alarm* n'est pas présente dans le modèle de produit.

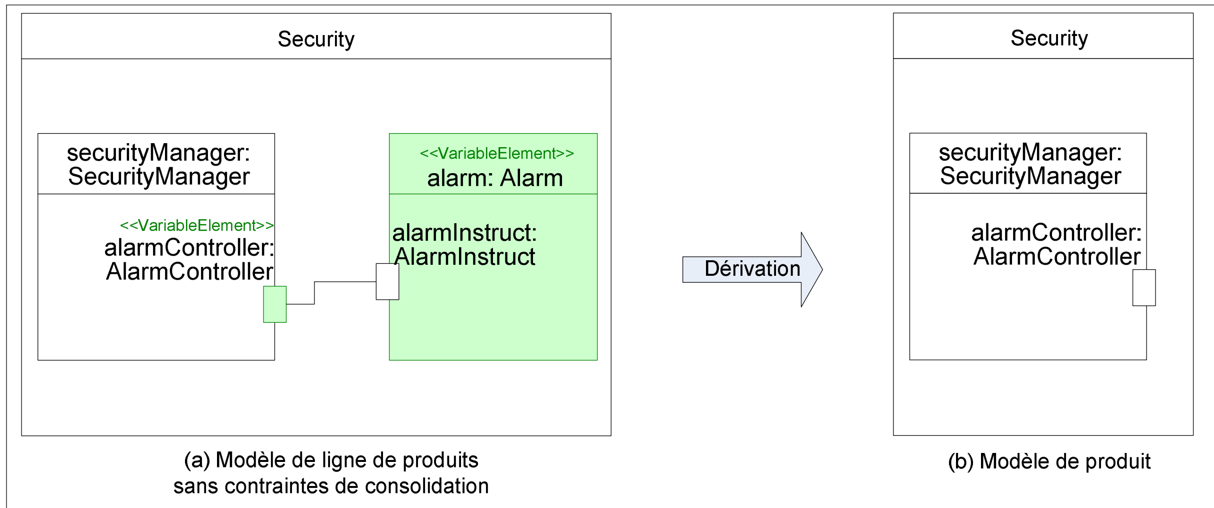


Figure 3.9 : Avant application de la règle de consolidation *Reg4*

Pour éviter ce type de problème, nous appliquons la règle de consolidation (*Reg4*) dont on peut observer les résultats sur la figure ci dessous. La figure 3.10 montre l'application de (*Reg4)* par ajout de la contrainte d'implication associée au port *alarmController* et à la part *alarm* (voir figure3.10.(a)). La contrainte C_9 impose la présence de la part *alarm* si le port *alarmController* est présent dans un modèle de produit. Cette contrainte écarte la possibilité d'obtenir le modèle de produit incomplet. Ainsi, le modèle de produit obtenu dans la figure 3.10.(b) est complet. Il contient la part *alarm* avec le port *alarmController* grâce à l'ajout de la contrainte structurelle C_9 comme le montre la figure 3.10.(a).

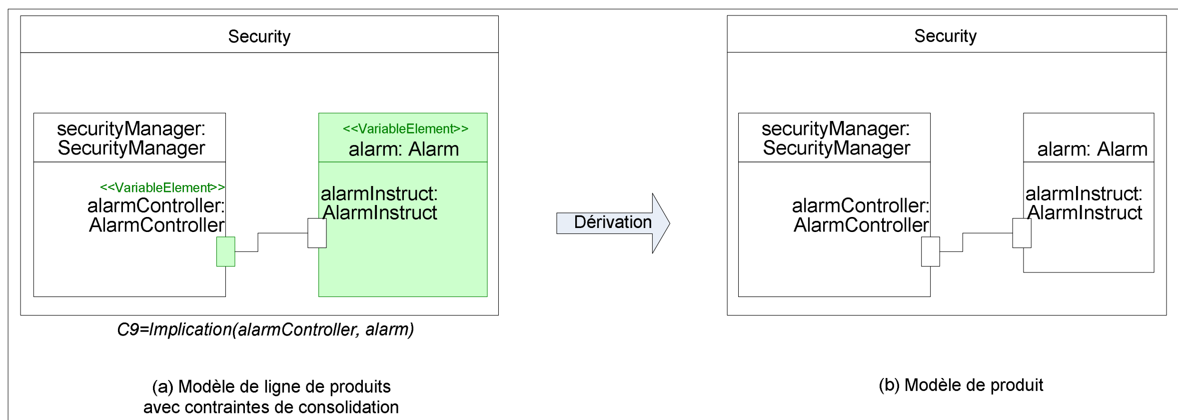


Figure 3.10 : Après application de la règle de consolidation *Reg4*

3.2.5. Règle de connecteur variable reliant deux ports simples (Reg5)

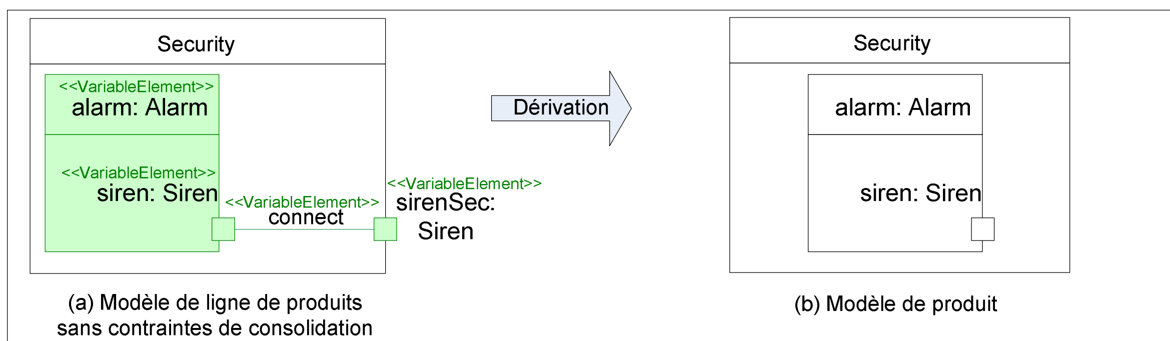
(Reg5) :

Si un connecteur variable relie deux ports dont au moins un est variable alors, le port obligatoire devient variable. Une contrainte de type équivalence doit être créée pour spécifier la coprésence du connecteur et des deux ports qu'il relie si un des trois est présent dans un modèle.

Exemple:

Nous reprenons l'exemple utilisé pour *(Reg1)* et nous supposons que le connecteur qui relie les deux ports simples est un connecteur variable nommé *connect*, comme indiqué sur la figue 3.11.(a).

Nous pouvons remarquer que les ports *siren* et *sirenSec* sont variables et qu'aucune contrainte structurelle n'est associée aux éléments du modèle, (voir la figure 3.11.(a)). La dérivation du modèle de ligne de produits, illustré dans 3.11.(a), permet d'obtenir le modèle de produit présenté dans la figure 3.11.(b). Nous pouvons bien constater que le modèle de produit obtenu est structurellement incomplet. Ceci s'explique par l'absence de contrainte structurelle qui relie les éléments *siren*, *connect* et *sirenSec*.



(a) Modèle de ligne de produits sans contraintes de consolidation

(b) Modèle de produit

Figure 3.11 : Avant application de la règle de consolidation *Reg5*

Pour remédier à ce problème, nous appliquons la règle de consolidation *(Reg5)* dont nous pouvons observer les résultats sur la figure suivante. La figure 3.12 montre l'application de *(Reg5)* par ajout de la contrainte d'équivalence associée aux éléments de connexion *siren*, *connect* et *sirenSec* (voir figure 3.12.(a)). La contrainte *C* impose la coprésence des deux ports et du connecteur, dans le cas où un des trois doit être présent dans le modèle de produit dérivé. Cette contrainte écarte la possibilité d'obtenir le modèle incomplet. Ainsi, le modèle de produit obtenu dans la figure 3.12.(b) est complet et contient les deux ports *siren* et *sirenSec* et le connecteur *connect* grâce à l'ajout de la contrainte structurelle *C* comme illustré dans la figure 3.12.(a).

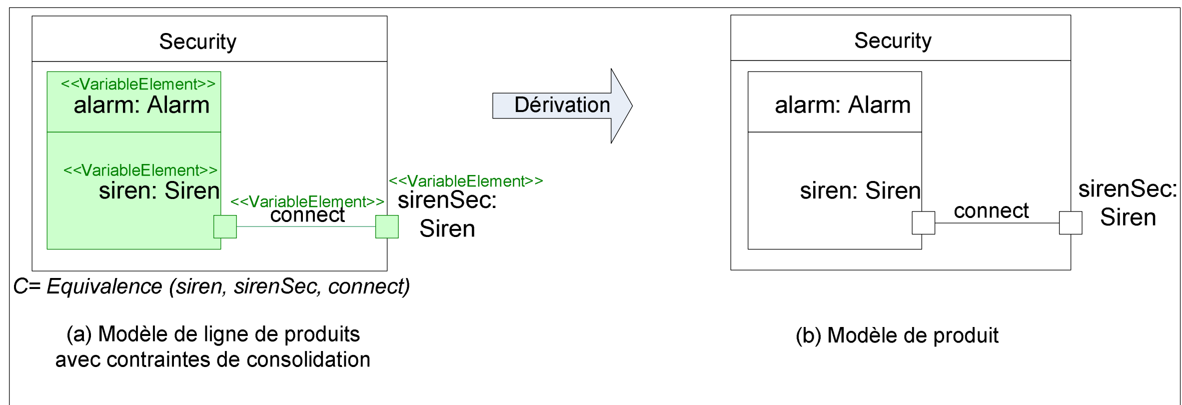


Figure 3.12 : Après application de la règle de consolidation *Reg5*

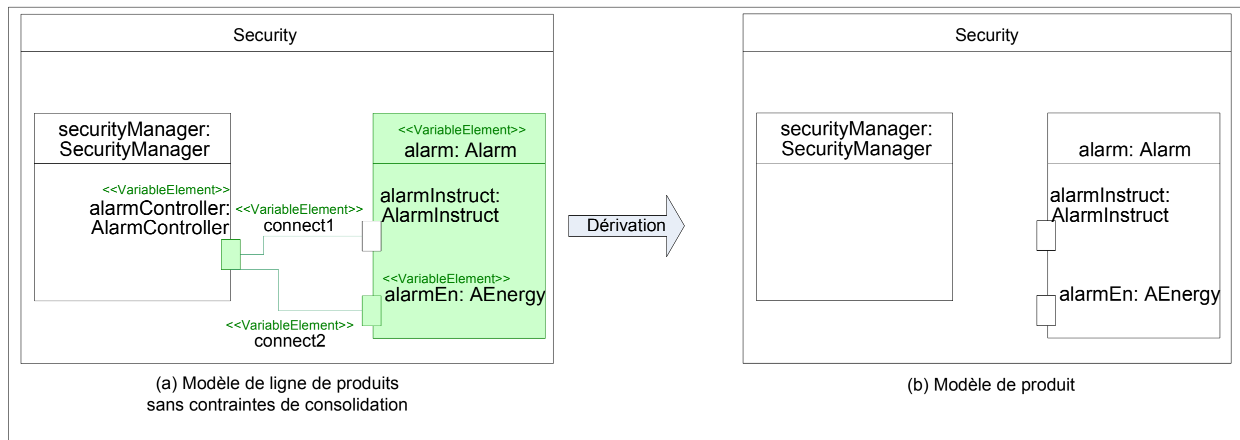
3.2.6. Règle de connecteur variable reliant deux ports dont au moins un est multiple (Reg6) ***(Reg6) :***

Si un port est connecté à un autre port multiple via un connecteur variable et qu'au moins un des deux ports est variable, alors le deuxième le devient aussi. Une contrainte de type implication doit être créée pour spécifier que la présence du connecteur dans un modèle de produit implique la présence du premier port et que la présence de ce dernier implique la présence du port multiple.

Exemple:

Nous reprenons l'exemple utilisé pour *(Reg2)* et nous supposons que les connecteurs *connect1* et *connect2* qui relient deux ports sont variables, comme indiqué sur la figue 3.13.(a).

Dans la figure 3.13.(a), le port *alarmController* attaché sur la part *securityManager* est un port multiple connecté aux deux ports *alarmInstruct* et *alarmEn* qui sont de leur côté attachés sur la part *alarm*. Nous pouvons remarquer que les éléments *alarmEn*, *connect2* et *alarmController* sont variables et qu'aucune contrainte structurelle n'est associée aux éléments du modèle, (voir la figure 3.13.(a)). La dérivation du modèle de ligne de produits, illustré dans 3.13.(a), montre l'obtention du modèle de produit présenté dans la figure 3.13.(b). Nous pouvons bien constater que le modèle de produit obtenu est structurellement incomplet. En effet, les ports *alarmInstruct* et *alarmEn* sont présents afin d'exposer les services de la part *alarm*. Cependant, le port *alarmController* ainsi que les connecteurs *connect1* et *connect2* ne sont pas présents dans le modèle de produit. Ceci s'explique par l'absence de contrainte structurelle qui relie les connecteurs *connect1* et *connect2* aux ports *alarmInstruct*, *alarmEn* et *alarmController*.

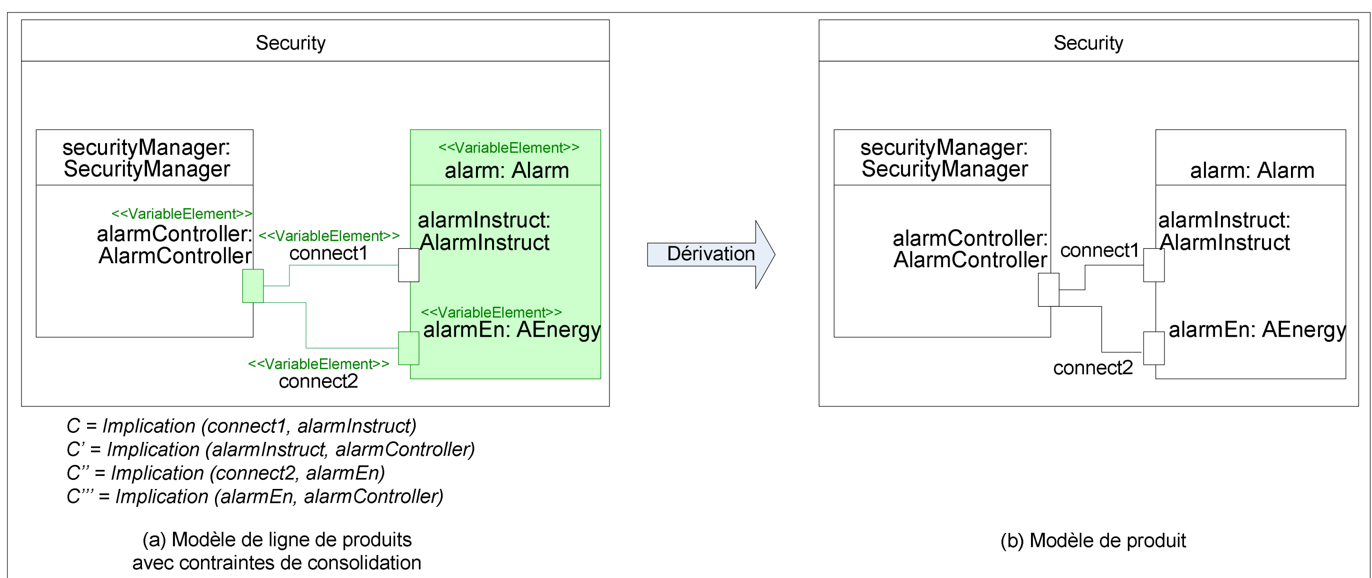


(a) Modèle de ligne de produits sans contraintes de consolidation

(b) Modèle de produit

Figure 3.13 : Avant application de la règle de consolidation *Reg6*

Pour éviter ce type de problème, nous appliquons la règle de consolidation *(Reg6)* dont les résultats s'affichent sur la figure ci dessous. La figure 3.14 montre l'application de *(Reg6)* par ajout des contraintes *C*, *C'*, *C''* et *C'''* (voir figure 3.14.(a)). La contrainte *C'* impose la présence du port *alarmController* si le port *alarmInstruct* est présent dans un modèle de produit. La présence de ce dernier port est à son tour conditionnée par celle du connecteur *connect1*, selon la contrainte *C*. La contrainte *C'''* impose la présence du port *alarmController* si le port *alarmEn* est présent dans un modèle de produit. La présence de ce dernier port est à son tour conditionnée par celle du connecteur *connect2*. Le modèle de produit obtenu dans la figure 3.14.(b) est complet grâce à l'ajout des contraintes structurelles *C*, *C'*, *C''* et *C'''*.



(a) Modèle de ligne de produits avec contraintes de consolidation

(b) Modèle de produit

Figure 3.14 : Après application de la règle de consolidation *Reg6*

Suite à l'explication des règles de consolidation des modèles de lignes de produits, nous pouvons bien constater leur utilité et leur importance dans l'obtention de modèles de produit structurellement cohérents et complets. Nous retrouvons l'exemple de la ligne de produits de sécurité «Détecteur de présence avec alarme» sur lequel nous allons appliquer les règles de consolidation.

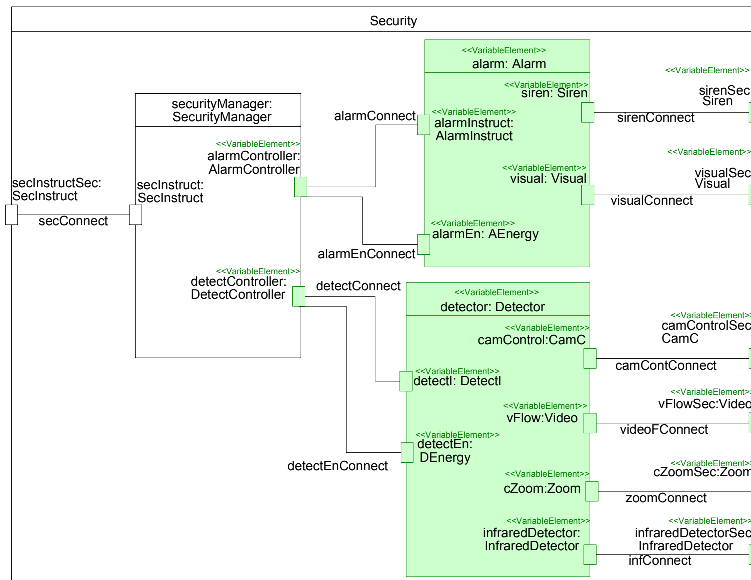


Figure 3.15 : Modèle de la ligne de produits Security après consolidation

L'application des règles de consolidation se traduit par la modification de l'information de variabilité de certains éléments du modèle et par l'ajout de nouvelles contraintes structurelles associées aux éléments variables. La figure 3.15 montre les exemples d'*alarmInstruct* et de *detectI* qui sont devenus variables. L'application des règles de consolidation dans ce cas d'exemple implique l'ajout des contraintes suivantes :

- Par application de *(Reg1)* les contraintes ajoutées sont :

$C_5 = Equivalence\ (siren,\ sirenSec)$

$C_{10} = Equivalence\ (visual,\ visualSec)$

$C_{11} = Equivalence\ (camControl,\ camControlSec)$

$C_{12} = Equivalence\ (vFlow,\ vFlowSec)$

$C_{13} = Equivalence\ (cZoom,\ cZoomSec)$

$C_{14} = Equivalence\ (infraredDetector,\ infraredDetectorSec)$

- Par application de *(Reg2)* les contraintes ajoutées sont :

  C_6 = *Implication (alarmInstruct, alarmController)*

  C_7 = *Implication (alarmEn, alarmController)*

  C_{15} = *Implication (detectI, alarmController)*

  C_{16} = *Implication (detectEn, alarmController)*

- Par application de *(Reg3)* les contraintes ajoutées sont :

  C_8= *Implication (siren, alarm)*

  C_{17} = *Implication (visual, alarm)*

  C_{18} = *Implication (alarmInstruct, alarm)*

  C_{19} = *Implication (alarmEn, alarm)*

  C_{20} = *Implication (camControl, detector)*

  C_{21} = *Implication (vFlow, detector)*

  C_{22} = *Implication (cZoom, detector)*

  C_{23} = *Implication (infraredDetector, detector)*

  C_{24} = *Implication (detectI, detector)*

  C_{25} = *Implication (detectEn, detector)*

- Par application de *(Reg4)* les contraintes ajoutées sont :

  C_9 = *Implication (alarmController, alarm)*

  C_{26} = *Implication (detectController, detector)*

L'application de *(Reg4)* sur les ports *camControlSec*, *vFlowSec*, *cZoomSec* et *infraredDetectroSec* attachés sur la part *detector* et celle sur les ports *sirenSec* et *visualSec* attachés sur la part *alarm* impliquent une redondance de contraintes, ce qui n'est pas souhaitable. Par exemple les contraintes C_5 et C_8 correspondent à l'application de *(Reg4)* pour

sirenSec. L'application des règles *(Reg5)* et *(Reg6)* ne génèrent pas de contraintes puisqu'il n'y a pas de connecteurs variables.

3.2.7. Synthèse

Dans cette section, nous avons montré que les contraintes des utilisateurs concernant les options d'une ligne de produits ne sont pas suffisantes pour obtenir des modèles de produit structurellement complets. C'est la raison pour laquelle nous avons proposé un ensemble de règles de consolidation du modèle de ligne de produits logiciels. Ces règles ont été élaborées en se basant sur des règles de cohérence extraites du méta-modèle UML ainsi que des règles métier. Nous avons illustré l'utilisation de ces règles de consolidation sur l'exemple de ligne de produits de sécurité «Détecteur de présence avec alarme». L'application de ces règles se matérialise par la modification de l'information de variabilité de certains éléments structurels, ainsi que par l'ajout de contraintes structurelles associées aux éléments variables du modèle. L'utilisation de règles de consolidation qui génèrent des contraintes a pour avantage d'assurer l'indépendance du modèle de ligne de produits logiciels. En effet, ces contraintes peuvent bien être modifiées ou évoluées sans impliquer des changements conséquents sur le modèle de ligne de produits. L'application des règles de consolidation produit des modèles de produit structurellement cohérents et complets. De plus, l'application automatique de ces règles évite au concepteur de la ligne de produits de vérifier la variabilité induite par la structure en plus de la variabilité spécifiée par l'utilisateur.

Dans la section suivante, nous présenterons deux mécanismes pour la composition des modèles de lignes de produits logiciels.

3.3. Formes de composition

La composition de modèles de lignes de produits logiciels permet d'intégrer des fragments de modèles développés séparément par plusieurs intervenants. Elle permet aussi la réutilisation des modèles existants pour obtenir de nouveaux modèles [61]. Dans notre contribution, nous nous sommes basés sur le rôle des diagrammes de structures composites d'UML dans la réutilisation de modèles ainsi que sur nos connaissances dans le domaine de lignes de produits. Nous nous sommes aussi inspirés des travaux existants dans la bibliographie comme par exemple les approches de composition des modèles de caractéristiques et celles qui se basent sur la modélisation orientée aspect.

Nous proposons deux formes de composition des modèles de lignes de produits logiciels, illustrées dans la figure 3.16 : la fusion et l'agrégation.

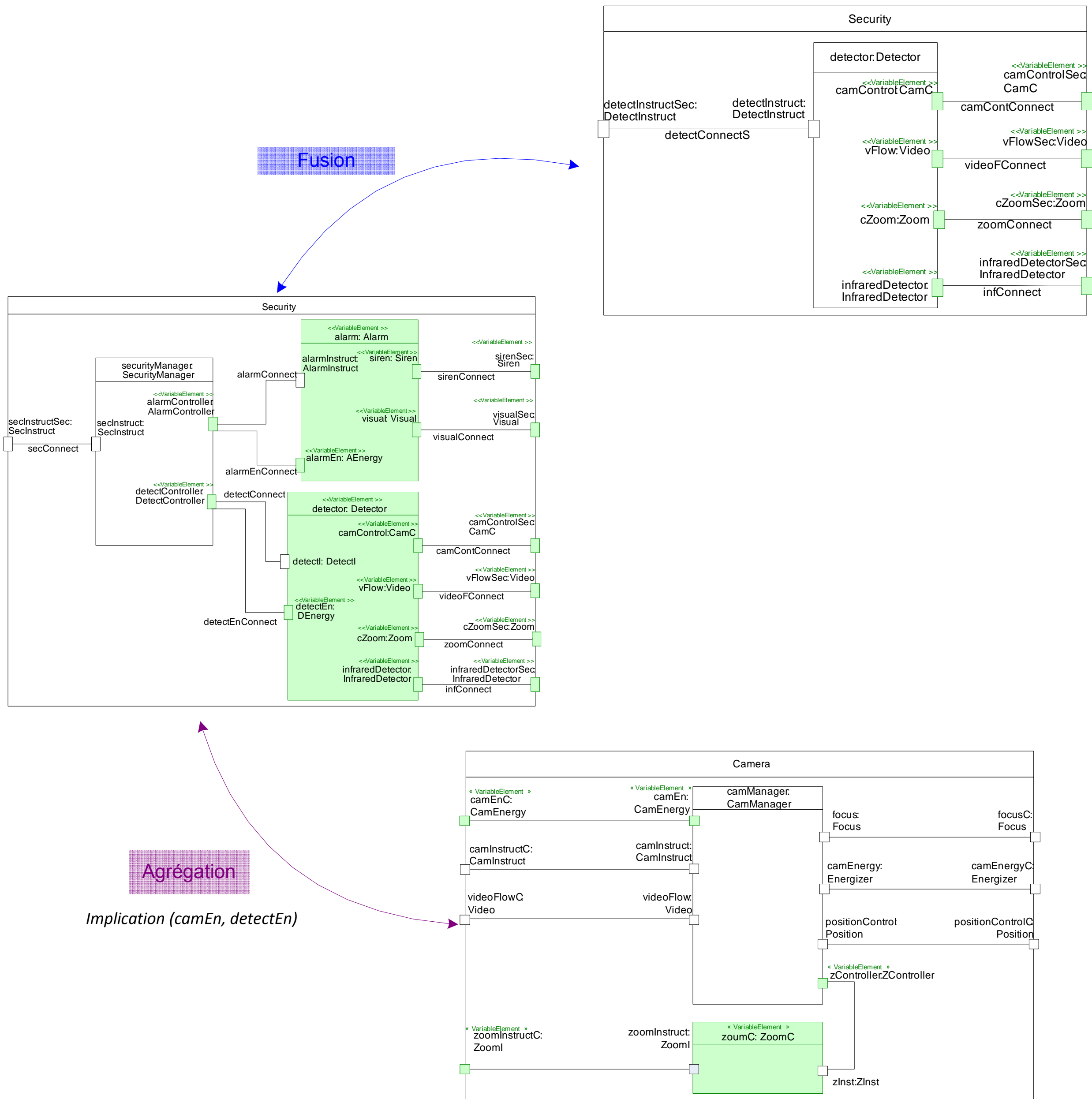


Figure 3.16 : Exemple de fusion et agrégation de modèles de lignes de produits

La première forme de composition, appelée fusion, réalise la combinaison des modèles de lignes de produits logiciels incluant des éléments structurels communs. Ces modèles représentent des fragments partiels d'un modèle global de ligne de produits logiciels que nous souhaitons obtenir en réalisant la fusion. La figure 3.16 présente l'exemple de deux modèles de ligne de produits ayant été élaborés par différents intervenants pour décrire une famille de produits de sécurité. L'exemple illustré montre que les deux classes composites nommées *Security* présentent certaines similarités au niveau de leurs éléments structurels. Par exemple, la part *detector* est un élément commun aux deux modèles. De même pour les ports *camControlConnect*, *camControlConnectSec*, *vFlow*, *vFlowSec*, *cZoom*, *cZoomSec*, *infraredDetector* et *infraredDetectorSec* qui sont aussi des éléments communs aux deux modèles.

La deuxième forme de composition, appelée agrégation, supporte la combinaison de modèles de lignes de produits logiciels n'ayant pas de similarités et dont les éléments structurels peuvent être liés par des contraintes transversales. La figure 3.16 présente l'exemple d'agrégation de deux modèles de lignes de produits, *Security* et *Camera*. Comme nous pouvons le constater sur la figure, ces deux modèles ne présentent pas de similarités. Ils sont reliés via une contrainte transversale associée à leurs éléments variables respectifs *detectEn* et *camEn*.

3.4. Conclusion

Nous avons vu dans ce chapitre que le choix de représenter une ligne de produits logiciels par un diagramme de structures composites suscite le besoin de consolider ce modèle du fait qu'il contient des éléments structurels variables. Pour assurer que le modèle de ligne de produits puisse permettre l'obtention de modèles de produit structurellement complets, nous avons proposé un ensemble de règles de consolidation de modèles. Ces règles se matérialisent par la modification de l'information de variabilité de certains éléments structurels ainsi que par l'ajout de contraintes structurelles. Les contraintes permettent de restreindre l'ensemble des modèles de produit obtenus à l'ensemble des modèles de produit structurellement complets. Nous avons ensuite exposé une vue globale des contributions qui seront proposées dans la suite de ce manuscrit. Deux formes de composition sont identifiées et définies à savoir la fusion et l'agrégation. Dans le chapitre suivant, nous détaillerons la démarche de la fusion dans laquelle la consolidation des modèles de lignes de produits logiciels constituera une étape majeure.

CHAPITRE 4
Fusion des modèles de lignes de produits logiciels

4.1. Exemple illustratif 60
4.2. Démarche de fusion : aperçu global 62
4 .3. Consolidation des modèles en entrée : première phase de fusion 63
4.4. Fusion des modèles en entrée : deuxième phase de fusion 65
4.4.1. Etape de comparaison 67
4.4.2. Etape de fusion 71
4.4.2.1. Propriétés sémantiques de fusion 71
4.4.2.2. Fusion des éléments structurels 72
4.4.2.3. Fusion des contraintes de variabilité 78
4.4.3. Synthèse 82
4.5. Consolidation du modèle de fusion résultant : troisième phase de fusion 83
4.6. Conclusion 85

Ce chapitre a pour objectif de présenter la démarche proposée pour la fusion des modèles de lignes de produits logiciels. Il est structuré en cinq sections. La première section présente un exemple illustratif qui sera utilisé tout au long de la démarche de fusion. La deuxième section donne un aperçu global de la démarche. La troisième section présente la première phase responsable de la consolidation des modèles à fusionner. La quatrième section détaille les traitements effectués pour la fusion des modèles d'entrée consolidés. Enfin, la cinquième section présente la troisième phase chargée de consolider le modèle de fusion résultant.

4.1. Exemple illustratif

Nous poursuivons avec l'exemple de fusion présenté dans la figure 3.16 du chapitre précédent. Il s'agit de deux modèles de ligne de produits de sécurité :

- Modèle de ligne de produits de sécurité « Détecteur de présence » : l'exemple une ligne de produits est celui qui détecte des présences suspectes de deux manières différentes; via une caméra surveillance qui enregistre une vidéo ou via un détecteur de mouvement infrarouge. La figure 4.1 représente un modèle de la structure composite de la ligne de produits *Security*. Ce modèle considère que la ligne *Security* contient un détecteur fixe, modélisé par une part nommée *detector*, qui gère la sécurité via deux options différentes. Nous modélisons la première option par les ports *camControl*, *camControlSec, VFlow*, *VFlowSec, Czoom* et *CzoomSec* dénotés comme variables. La deuxième option, quant à elle, est modélisée sous forme de ports variables *infraredDetector* et *infraredDetectorSec*. La ligne de produits S*ecurity*, comme l'illustre la figure 4.1, expose ses services via le port fixe *detecInstructSec*. Les connecteurs comme par exemple *camContConnect*, *videoConnect* et *zoomConnect,* réalisent la connexion à travers les ports.

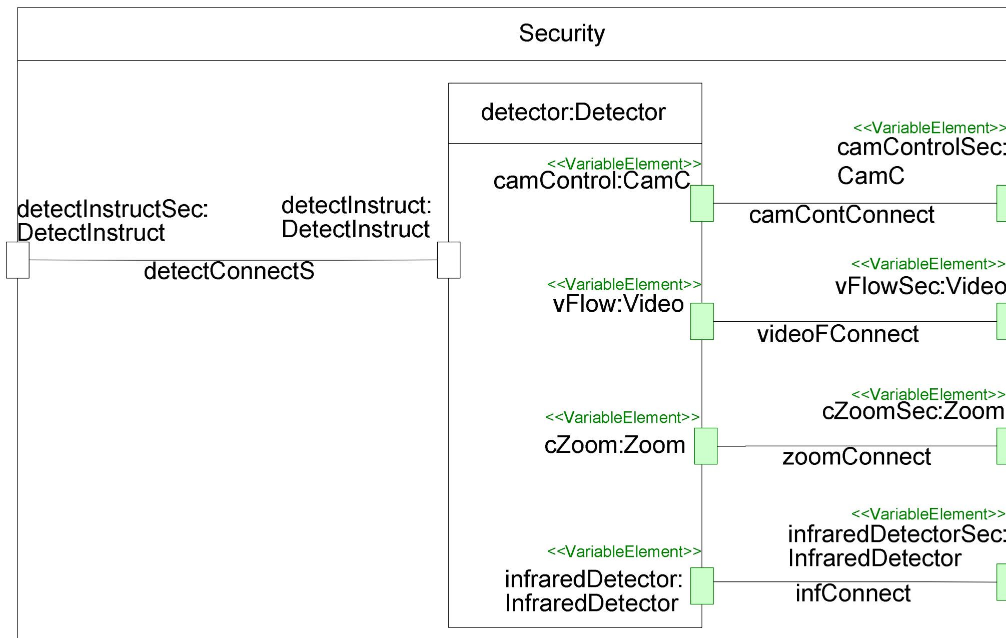


Figure 4.1 : Modèle de la ligne de produits de sécurité « Détecteur de présence »

Afin qu'ils soient présents dans un modèle de produit valide, le choix des éléments variables doit respecter les contraintes suivantes :

C_I= *AtLeastOne (camControl, infraredDetector),* C_I est la contrainte qui relie les deux ports variables *camControl* et *infraredDetector*. *AtLeastOne* est le type de la contrainte C_I. Cette contrainte indique qu'un modèle de produit doit inclure au moins l'option *camControl* ou *infraredDetector* ou bien les deux.

C_{II}= *Equivalence (camControl, vFlow, cZoom),* C_{II} est la contrainte qui relie les trois ports variables *camControl*, *vFlow* et *cZoom*. *Equivalence* est le type de la contrainte C_{II}. Cette contrainte indique qu'un modèle de produit doit inclure obligatoirement les trois ports *camControl*, *vFlow* et *cZoom* si l'un des trois est présent.

- Modèle de ligne de produits de sécurité « Détecteur de présence avec alarme » : la figure 4.2 présente le modèle de ligne de produits de sécurité décrit dans le chapitre précédent (voir la page 40).

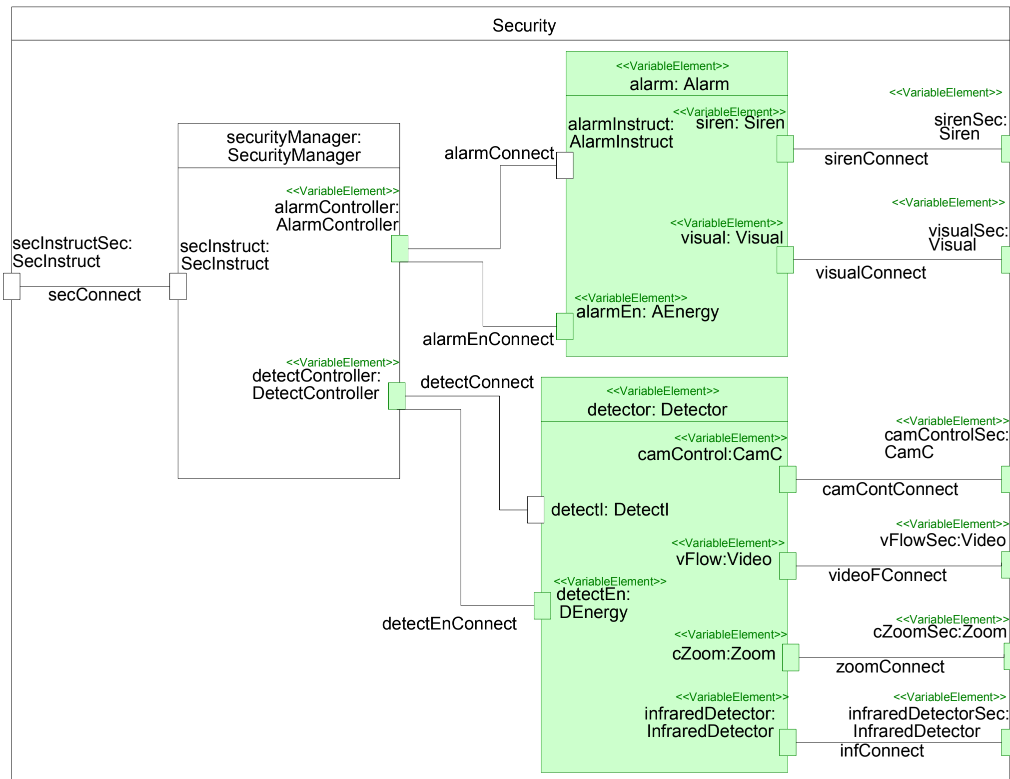


Figure 4.2: Modèle de la ligne de produits de sécurité « Détecteur de présence avec alarme»

Les deux modèles présentés ci-dessus servirons d'exemples pour l'illustration de la démarche de fusion proposée dans la suite.

4.2. Démarche de fusion : aperçu global

La figure 4.3 donne un aperçu global sur la démarche de fusion proposée. Nous distinguons trois principales phases : la première phase, appelée phase de consolidation, a pour objectif d'assurer que les modèles en entrée soient structurellement cohérents et complets. Nous utilisons pour cet objectif un ensemble de règles de consolidation, qui une fois appliqué aux modèles en entrée, permet d'obtenir des modèles consolidés et prêts à être fusionnés. La deuxième phase, nommée phase de fusion, met en œuvre les différents traitements effectués pour la fusion des modèles de lignes de produits. A l'issue de cette phase, nous obtenons un modèle fusionné qui doit passer par une seconde phase de consolidation au court de laquelle les mêmes règles de consolidation seront à nouveau appliquées afin d'aboutir à un modèle de fusion consolidé.

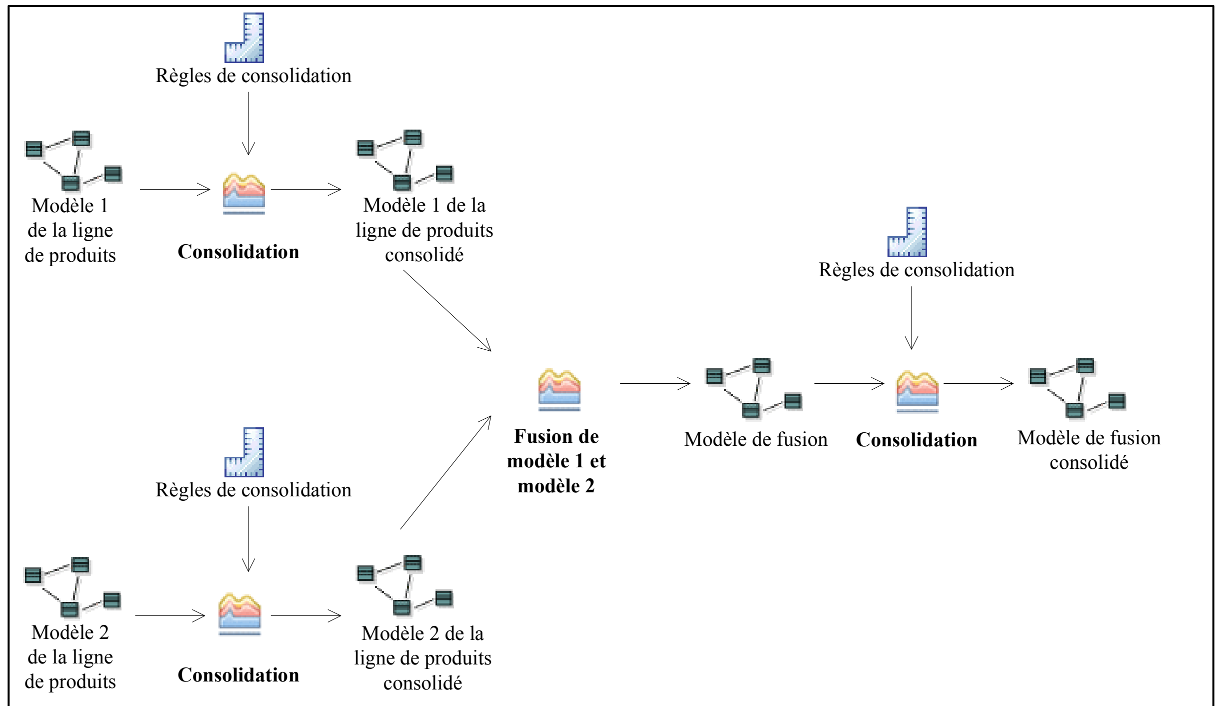


Figure 4. 3 : Vue globale de la démarche de fusion

Suite à la description de la vue globale de la démarche de fusion, nous détaillerons les différentes phases qui la constituent dans la suite.

4 .3. Consolidation des modèles en entrée : première phase de fusion

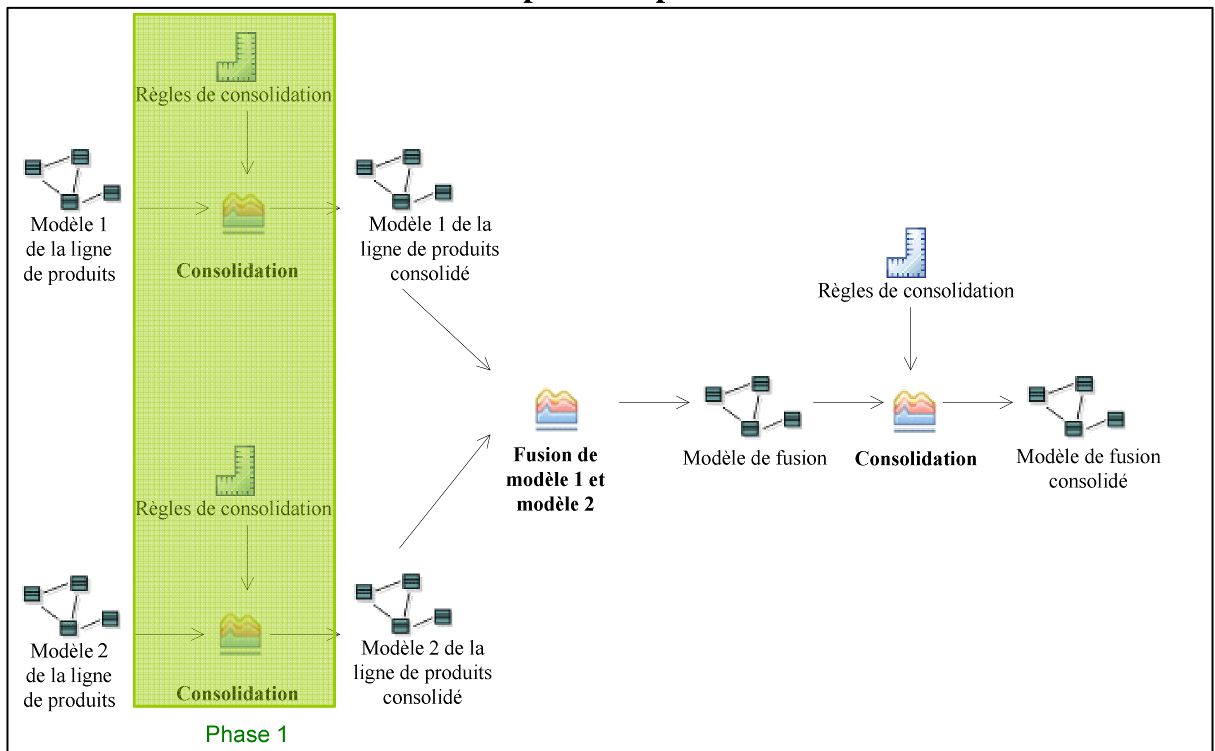


Figure 4.4 : La phase de consolidation

La phase de consolidation des modèles en entrée représente la première phase de la démarche globale de fusion, voir figure 4.4.

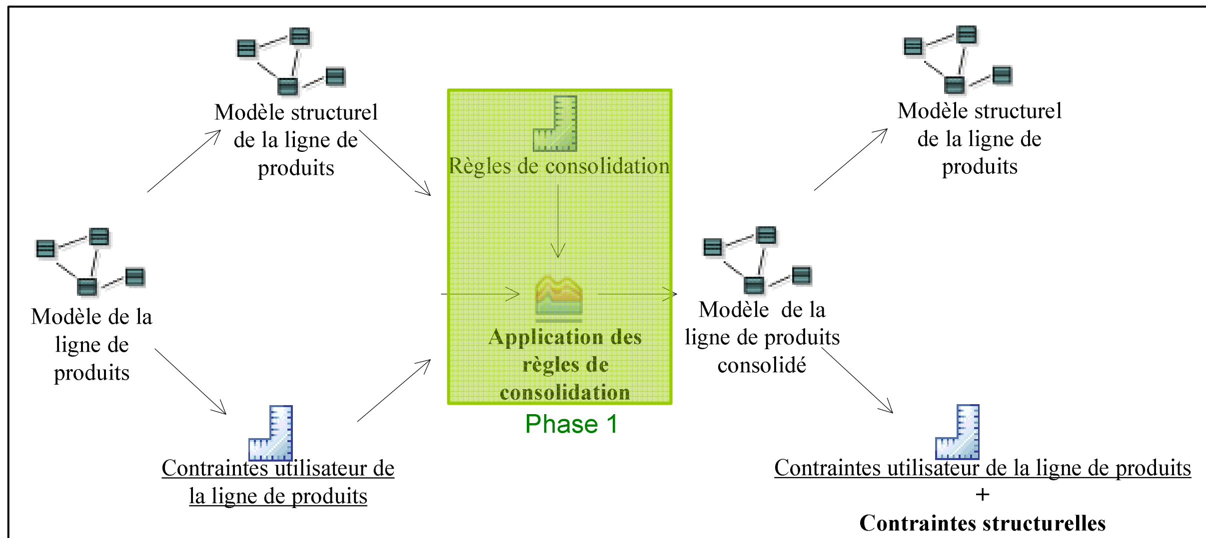


Figure 4.5 : Vue détaillée de la phase de consolidation

Cette phase consiste à consolider les modèles à fusionner. Pour chacun des modèles de lignes de produits en entrée, la phase de consolidation consiste à appliquer l'ensemble des règles définies dans la section 3.2 du chapitre précédent et dont la conséquence peut se matérialiser par la modification de l'information de variabilité (c'est-à-dire obligatoire, variable ou absent) de certains éléments structurels ainsi que par l'ajout d'un nouvel ensemble de contraintes structurelles, voir figure 4.5. Le but de ces règles est d'assurer au concepteur l'obtention de modèles de produit structurellement cohérents et complets.

L'application des règles de consolidation sur les exemples de modèles de ligne de produits de sécurité illustrés dans les figures 4.1 et 4.2 permet d'avoir les résultats suivants :

- Pour le modèle de ligne de produits de sécurité « Détecteur de présence », l'application des règles de consolidation permet d'obtenir le modèle consolidé avec l'ensemble des contraintes structurelles suivantes :

Par application de *(Reg1)* les contraintes ajoutées sont :

$C_{III} = Equivalence\ (camControl,\ camControlSec)$

$C_{IV} = Equivalence\ (vFlow,\ vFlowSec)$

$C_{V} = Equivalence\ (cZoom,\ cZoomSec)$

$C_{VI} = Equivalence\ (infraredDetector,\ infraredDetectorSec)$

- Pour le modèle de ligne de produits sécurité « Détecteur de présence avec alarme» les contraintes engendrées par l'application des règles de consolidation ont été présentées dans la section 3.1.2.

4.4. Fusion des modèles en entrée : deuxième phase de fusion

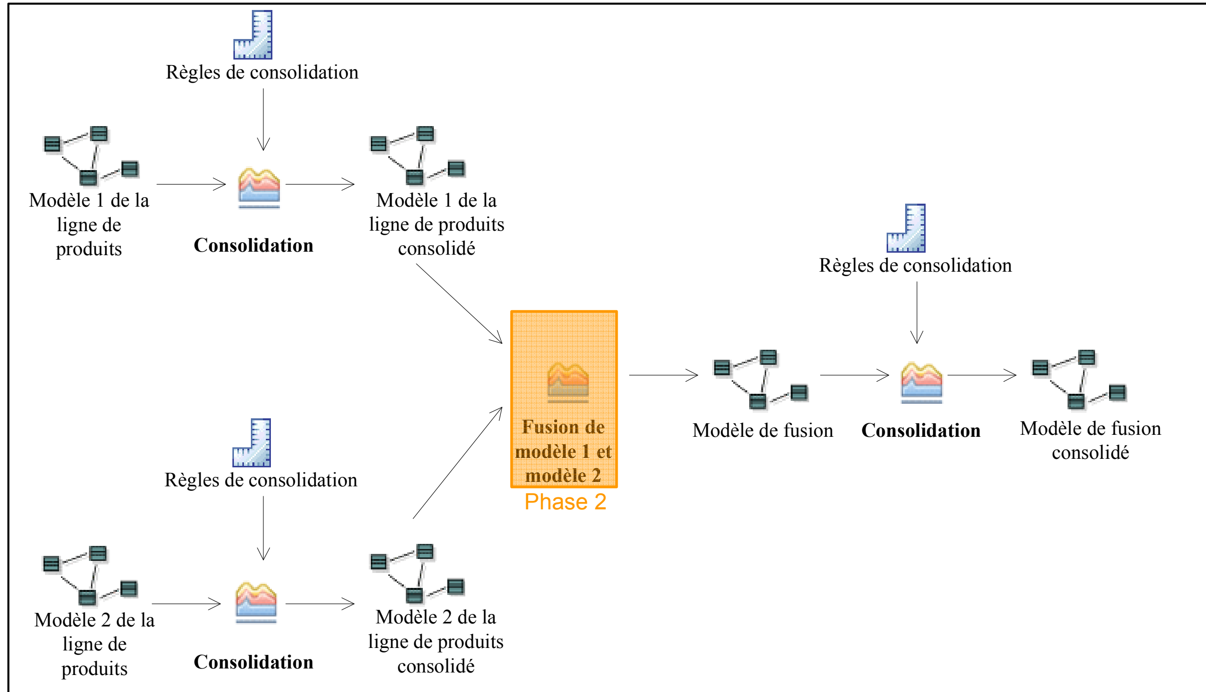


Figure 4.6 : La phase de fusion

La phase de fusion des modèles en entrée représente la deuxième phase de la démarche globale de fusion, voir figure 4.6.

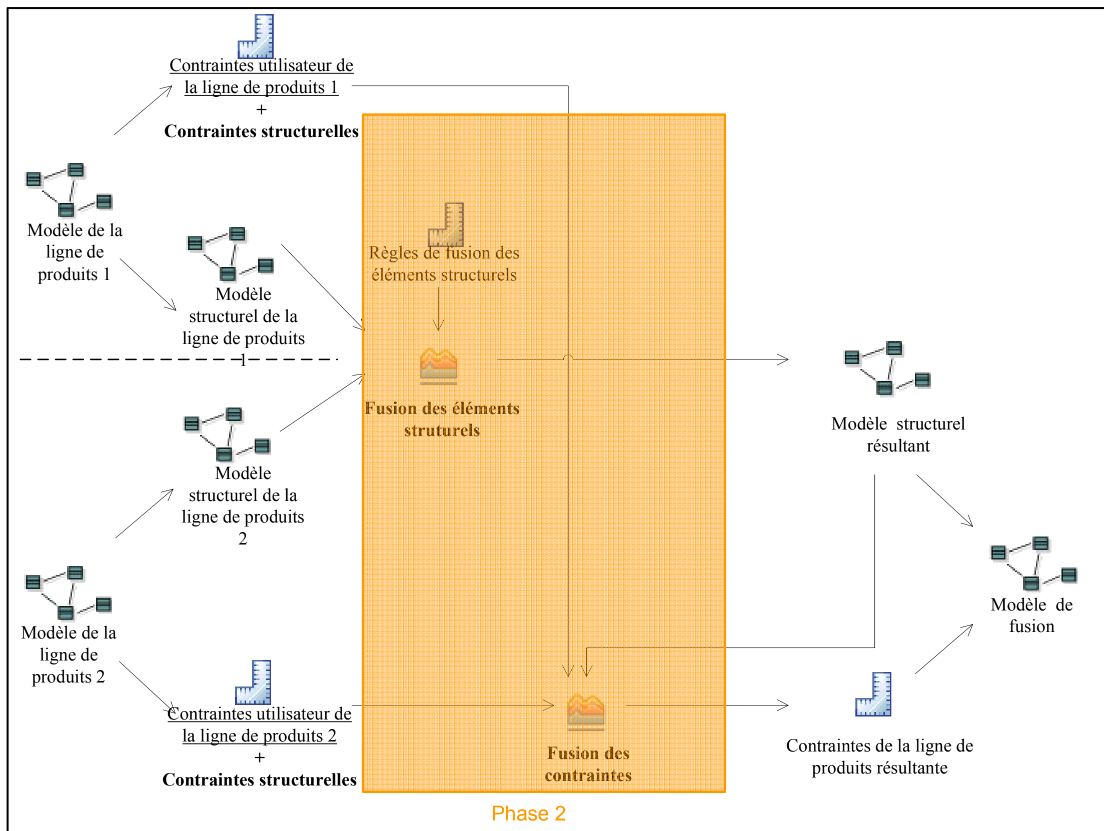


Figure 4.7 : Vue détaillée de la phase de fusion

L'objectif de cette phase est de combiner les parties chevauchantes (communes) des modèles en entrée afin d'obtenir le modèle de ligne de produits global c'est à dire le modèle de fusion. La figure 4.7 montre que cette phase s'intéresse non seulement à la fusion des éléments structurels appartenant aux modèles d'entrée mais aussi à la fusion des contraintes de variabilité associées à ces éléments.

Nous proposons de réaliser la fusion des modèles en mode union telle que les systèmes valides qui peuvent être produits à partir du modèle de fusion soient des produits valides du premier ou du second modèle d'entrée.

La phase de fusion repose sur un ensemble d'hypothèses :

H1 : Préservation de la sémantique : La projection de la sémantique du modèle fusionné sur les éléments de modèles de départ est égale à la sémantique respective de ces modèles. Il s'agit bien d'une fusion en mode union.

La fusion en mode union permet de conserver l'ensemble des produits obtenus à partir de chacun des modèles de départ. D'autres modes sont envisageables comme la fusion en mode intersection. Cependant, dans cette thèse notre contribution s'intéresse uniquement à la fusion en mode union. Le travail présent pourrait bien être étendu pour inclure le mode intersection qui est aussi intéressant pour la fusion des modèles de lignes de produits logiciels.

H2 : La comparaison et la fusion ne s'appliquent qu'aux éléments structurels

H3 : Les éléments communs ont le même nom.

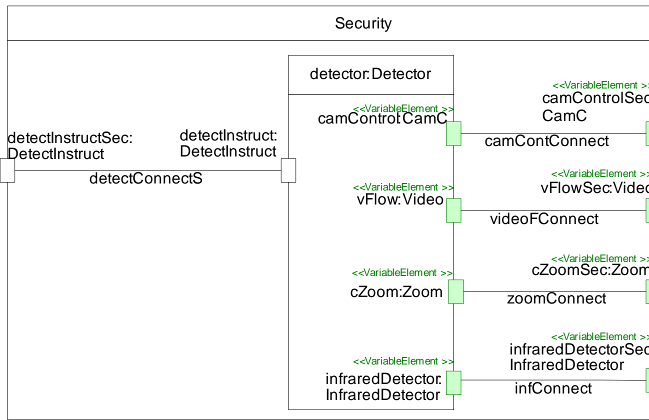
Pour réaliser cette phase de fusion, nous proposons deux étapes :
L'étape de comparaison (ou correspondance): Cette étape compare les éléments structurels des modèles en entrée et identifie les éléments qui représentent le même concept, c'est-à-dire les éléments communs entre les modèles en entrée. Le résultat de cette comparaison est utilisé dans la suite pendant l'étape de fusion, afin que les éléments communs représentant un même concept soient fusionnés pour former un élément intègre de ce concept.

L'étape de fusion : Cette étape réalise la fusion des éléments correspondants précédemment identifiés. De nouveaux éléments structurels intègres sont créés pour représenter les concepts. Les éléments résultants sont alors inclus dans le modèle de fusion. Cette étape comprend aussi la fusion des contraintes des modèles considérés en entrée.

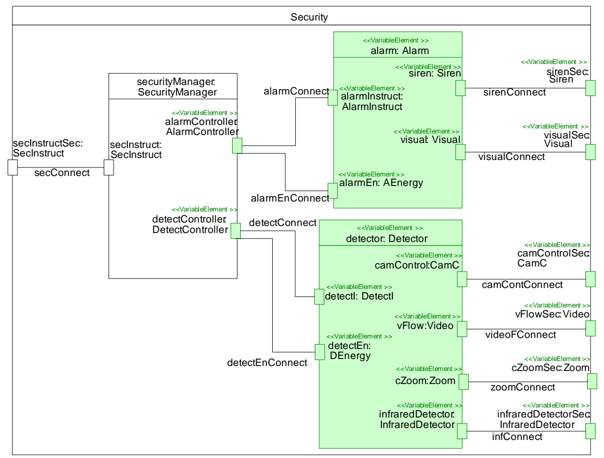
4.4.1. Etape de comparaison

Deux éléments de modèles en entrée correspondent s'ils décrivent le même concept. Nous nous basons sur la comparaison de signatures d'éléments afin de fournir plus de précisions aux critères de correspondance. Chaque élément appartenant aux modèles d'entrée possède une signature qui consiste en son nom et sa propriété type. La signature d'un élément peut être écrite de la manière suivante : *Signature = {Nom, pType}*

La propriété type d'un élément est définie par son nom et sa méta-classe et peut être écrite de la manière suivante : *pType = {Nom, Meta-classe}*



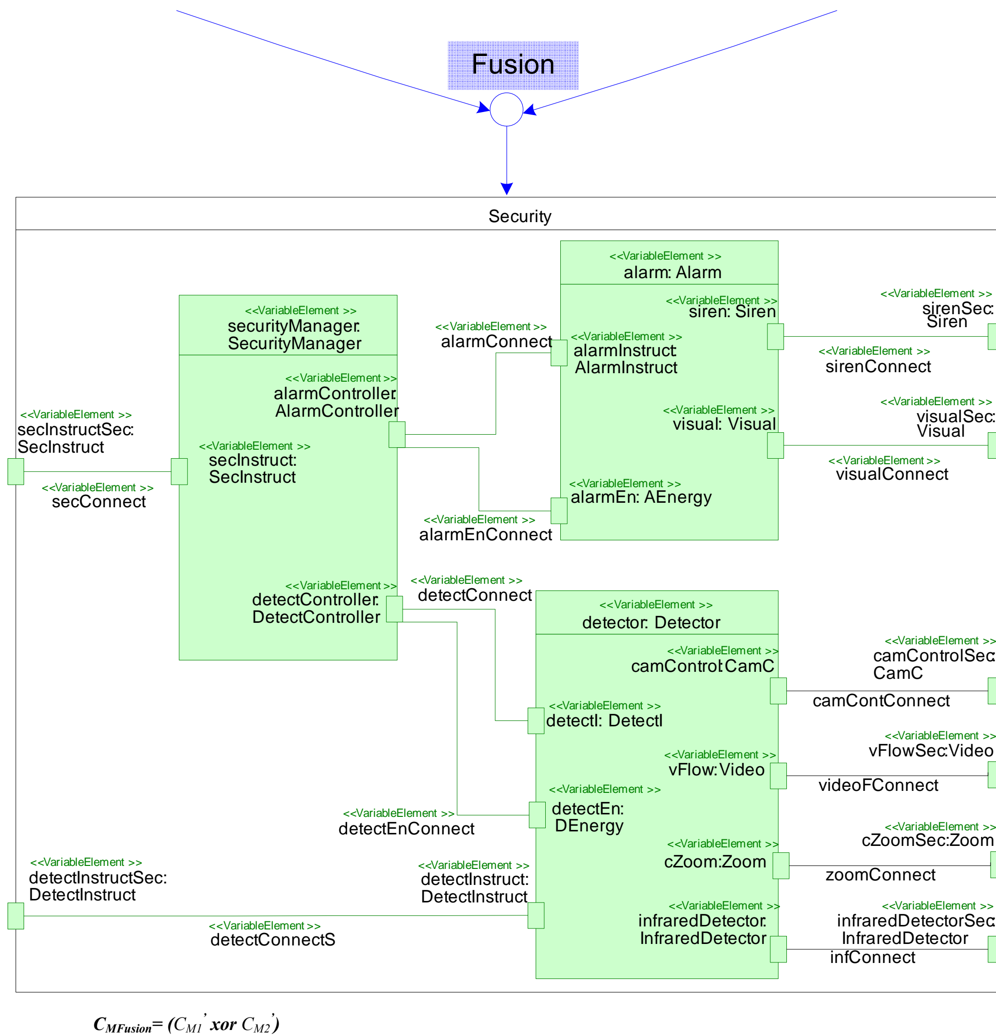
C_{M1}= (C_I and C_{II} and C_{III} and C_{IV} and C_V and C_{VI})



C_{M2}= (C_1 and C_2 and C_3 and C_4 and C_5 and C_6 and … and C_{26})

(a) Modèle 1 de la ligne de produits Security
« Détecteur de présence »

(b) Modèle 2 de la ligne de produits Security
« Détecteur de présence avec alarme »



$C_{MFusion}$= (C_{M1}' xor C_{M2}')

(c) Modèle global de ligne de produits Security résultat de fusion

Figure 4.8 : Exemple de fusion de modèles de lignes de produits

Considérons la figure 4.8. La signature de l'élément *alarm* dans la figure 4.8.(a) est *{Nom = alarm, pType = Alarm}*, où la propriété type d'*alarm* est définie par le couple *{Nom = Alarm, Meta-classe = Class}*. La comparaison des éléments se base sur les définitions suivantes :

(1) Deux éléments sont correspondants s'ils ont des signatures qui correspondent.

(2) Deux signatures d'éléments correspondent si elles possèdent le même nom d'éléments et que leurs propriétés type correspondent.

(3) Deux propriétés type correspondent quand elles ont le même nom et la même méta-classe. Dans le cas où leurs noms sont différents, elles ne peuvent être correspondantes que si elles sont des instances de la méta-classe *Class* d'UML et qu'elles possèdent le même super-type.

Les définitions *(1)*, *(2)* et *(3)* permettent de comparer les éléments UML de manière générale. Cependant, vu que notre contribution s'intéresse à la fusion des structures composites des lignes de produits, nous nous devons de définir avec précision les éléments structurels manipulés. Par conséquent, nous proposons dans la suite des définitions spécifiques à ces éléments en se basant sur les définitions précédemment présentées.

- ***Correspondance des parts UML :*** deux éléments UML e_1 et e_2 sont des parts correspondantes si e_1 et e_2 sont les parts de leurs conteneurs[1] respectifs, les signatures de e_1 et de e_2 correspondent ainsi que leurs conteneurs respectifs. *(4)*

Considérons l'exemple des éléments *detector* dans la figure 4.8. À l'image de l'élément *detector* de la figure 4.8.(a), qui représente une part de la classe composite *Security* du premier modèle, l'élément *detector* de la figure 4.8.(b) est une part de la classe composite *Security* du deuxième modèle. Les deux classes composites correspondent selon la définition *(3)*. De plus, dans la figure 4.8.(a) la signature de *detector* est définie par *{Nom = detector, pType = Detector}*, où la propriété type de *detector* est définie par le couple *{Nom = Detector, Meta-classe = Class}*. De même dans la figure 4.8.(b) la signature de *detector* est définie par *{Nom = detector, pType = Detector}*, où la propriété type de *detector* est définie par le couple *{Nom = Detector, Meta-classe = Class}*. Ainsi suivant la définition *(4)*, les éléments *detector* dans les figures 4.8.(a) et 4.8.(b) sont des parts correspondantes.

[1] Conteneur correspond à la propriété *owner* dans le méta-modèle UML. Elle indique l'élément qui contient un autre élément [11]

- ***Correspondance des ports UML*** **:** deux éléments UML e_1 et e_2 sont des ports correspondants s'ils représentent les ports de leurs conteneurs respectifs, leurs conteneurs correspondent ainsi que leurs signatures respectives. *(5)*

Considérons l'exemple des éléments *camControlSec* attachés sur les classes composites *Security* dans la figure 4.8. À l'image de l'élément *camControlSec* de la figure 4.8.(a), qui représente un port de la classe composite *Security* du premier modèle, l'élément *camControlSec* de la figure 4.8.(b) est un port de la classe composite *Security* du deuxième modèle. Les deux classes composites correspondent selon la définition *(3)*. De plus, dans la figure 4.8.(a) la signature de *camControlSec* est définie par *{Nom = camControlSec, pType = CamC}*, où la propiété type de *camControlSec* est définie par le couple *{Nom = CamC, Meta-classe = Class}*. De même dans la figure 4.8.(b) la signature de *camControlSec* est définie par *{Nom = camControlSec, pType = CamC}*, où la propriété type de *camControlSec* est définie par le couple *{Nom = CamC, Meta-classe = Class}*. Ainsi selon la définition *(5)*, les éléments *camControlSec* dans les figures 4.8.(a) et 4.8.(b) sont des ports correspondants.

- ***Correspondance des connecteurs UML*** **:** deux éléments UML e_1 et e_2 sont des connecteurs correspondants si :

- e_1 et e_2 sont les connecteurs de leurs conteneurs respectifs et correspondants, *(6.a)*
- les signatures de e_1 et e_2 correspondent ou ont le même nom et les valeurs de leurs propriétés type sont absentes[2], *(6.b)*
- pour chaque terminaison du connecteur e_1 et chaque terminaison du connecteur e_2 du même côté, les parts qui sont attachées à ces terminaisons sont correspondantes ou nulles, *(6.c)*
- pour chaque terminaison du connecteur e_1 et chaque terminaison du connecteur e_2 du même côté, les ports qui sont attachées à ces terminaisons sont correspondants ou nuls, *(6.d)*

Nous considérons l'exemple des éléments *camContConnect* dans la figure 4.8. La figure 4.8.(a) montre que *camContConnect* est un connecteur contenu dans la classe composite *Security*. De même, la figure 4.8.(b) montre que *camContConnect* est un connecteur contenu dans la classe composite *Security*. La définition *(6.a)* est donc vérifiée car les deux classes composites *Security* sont correspondantes. La signature de *camContConnect* de la figure

[2] La valeur de la propriété type est dite « absente » ou « nulle » si elle n'est pas affectée.

4.8.(a) est *{Nom = camContConnect, pType = nul}*. De même, la signature de *camContConnect* dans la figure 4.8.(b) est *{Nom = camContConnect, pType = nul}*. Les deux signatures correspondent (voir la définition *(2)*) et par conséquent la condition de correspondance établie par la définition *(6.b)* est remplie. D'autre part, dans la figure 4.8.(a) la terminaison gauche du connecteur *camContConnect* est attachée à la part *detector* ainsi qu'au port *camControl*. La terminaison droite de *camContConnect* n'est attachée à aucune part mais au port *camControlSec*. De même, dans la figure 4.8.(b) la terminaison gauche du connecteur *camContConnect* est attachée à la part *detector* ainsi qu'au port *camControl* alors que la terminaison droite de *camContConnect* n'est attachée à aucune part mais au port *camControlSec*. Vu que *detector*, *camControl* et *camControlSec* de la figure 4.8.(a) sont respectivement correspondants à *detector*, *camControl* et *camControlSec* de la figure 4.8.(b), nous déduisons que les conditions de correspondance spécifiées par les définitions (6.c) et (6.d) sont respectées. Par conséquent, les éléments *camContConnect* dans les figures 4.8.(a) et 4.8.(b) sont des connecteurs correspondants.

4.4.2. Etape de fusion

4.4.2.1. Propriétés sémantiques de fusion

La définition des propriétés sémantiques de la fusion est réalisée en fonction des paramètres suivants :

- La relation entre le modèle de ligne de produits global résultant de la fusion et les modèles d'entrée. *(Rel1)*
- La relation entre l'ensemble des systèmes valides obtenus à partir du modèle de ligne de produits résultant de la fusion et l'ensemble des systèmes valides obtenus à partir des modèles d'entrée. *(Rel2)*

Plus précisément, le résultat de la fusion doit satisfaire les propriétés suivantes :

- Les éléments obtenus dans le modèle de fusion résultant représentent la totalité des éléments contenus dans les modèles en entrée, sans redondances possibles. *(P1)*

Prenons l'exemple de la part *alarm* appartenant à la classe composite *Security* de la figure 4.8.(b) qui ne possède pas de correspondant dans la classe composite *Security* de la figure 4.8.(a). Elle est donc copiée dans le modèle de fusion résultant comme indiqué dans la figure 4.8.(c). D'autre part les parts *detector* dans la figure 4.8.(a) et *detector* dans la figure 4.8.(b) sont correspondants et le résultat de leur fusion , *detector*, est intégré dans le modèle de fusion comme le montre la figure 4.7.(c).

- Pour tout élément du modèle de fusion, la projection de son information de variabilité sur les éléments des modèles en entrée est égale à l'information de variabilité respective des éléments fusionnés. *(P2)*

Nous poursuivons avec l'exemple de *detector* dans la figure 4.8. La figure 4.8.(a) montre que *detector* est une part obligatoire, alors que la figue 4.8.(b) montre que *detector* est une part variable. Le résultat de leur fusion doit conserver leurs informations de variabilité c'est à dire obligatoire pour la première et variable pour la deuxième.

Remarquons que la conservation de l'information de variabilité comme définie par *(P2)* peut avoir un impact sur la relation *(Rel2)*. En effet, des systèmes qui ne sont valides pour aucun des modèles en entrée peuvent être obtenus à partir du modèle de fusion résultant. Pour éviter ce problème et conserver l'ensemble des systèmes valides obtenus à partir des modèles d'entrée, nous spécifions comment combiner les contraintes des modèles d'entrée en respectant la propriété suivante :

- Tout système valide obtenu du modèle de fusion résultant est un système valide obtenu à partir de l'un des modèles en entrée. *(P3)*

4.4.2.2. Fusion des éléments structurels

Nous proposons un catalogue de règles pour fusionner les éléments structurels contenus dans les modèles considérés en entrée. Ces règles de fusion ont pour objectif de décider de l'union ainsi que de l'information de variabilité des éléments résultants dans le modèle de fusion tel que les propriétés *(P1)* et *(P2)* soient vérifiées.

Ceci s'illustre pour les exemples de *detector* de la figure 4.8.(a) et *detector* de la figure 4.8.(b) qui correspondent. Le résultat de leur fusion, *detector*, est inclus dans le modèle de fusion comme le montre la figure 4.8.(c). Ainsi la propriété *(P1)* est respectée. D'autre part, *detector* dans la figure 4.8.(a) et *detector* dans la figure 4.8.(b) présentent des informations de variabilité différentes, le premier est un élément structurel obligatoire alors que le deuxième est un élément structurel variable. Considérant les informations de variabilité de ces éléments, le résultat de leur fusion doit exprimer sa propre information de variabilité dans le modèle de fusion résultant. Pour répondre à ce besoin, nous explicitons les formes que peut avoir l'information de variabilité associée aux éléments structurels et nous proposons un ensemble de règles de fusion qui respectent la propriété *(P2)*. Ces règles permettent de déterminer l'information de variabilité d'un élément résultant d'une fusion en considérant les informations de variabilité des éléments en entrée.

Model1/ Model2	Obligatoire	Variable	Absent
Obligatoire	Obligatoire	Variable	Variable
Variable	Variable	Variable	Variable
Absent	Variable	Variable	Absent

Tableau 4.1 : Fusion des éléments structurels- propriétés (P1) et (P2) sont respectées

Le tableau 4.1 présente trois formes de l'information de variabilité :

- Obligatoire : Ceci indique que l'élément structurel doit être présent dans tous les systèmes valides obtenus à partir de la ligne de produits. L'ensemble des possibilités pour un élément obligatoire peut être exprimé de la manière suivante : Obligatoire = {présent}
- Variable : Ceci indique que l'élément structurel peut être présent dans certains systèmes obtenus à partir de la ligne de produits, comme il peut être absent dans d'autres. L'ensemble des possibilités pour un élément variable peut être exprimé de la manière suivante : Variable = {présent, absent}
- Absent : Ceci indique que l'élément est absent dans le modèle de ligne de produits ainsi que dans tous les systèmes obtenus. L'ensemble des possibilités pour un élément absent peut être exprimé de la manière suivante : Absent = {absent}

Nous détaillons les règles de fusion présentées dans le tableau 4.1 et nous montrons comment elles respectent la propriété *(P2)* :

Obligatoire ○ Obligatoire = Obligatoire (R1)

- *(R1)* spécifie que si un élément obligatoire appartenant à un des modèles en entrée correspond à un élément obligatoire qui appartient à un autre modèle en entrée, alors l'élément résultant de leur fusion est un élément obligatoire qui appartient au modèle de fusion ((*P1*) est vérifiée). Nous détaillons *(R1)* comme suit:

  Obligatoire ○ Obligatoire = {présent} U {présent} = {présent}= Obligatoire. Par conséquent, la propriété *(P2)* est aussi vérifiée par *(R1)*.

Obligatoire ○ Variable = Variable (R2)

- *(R2)* spécifie que si un élément obligatoire appartenant à un des modèles en entrée correspond à un élément variable qui appartient à un autre modèle en entrée, alors l'élément résultant de leur fusion est un élément variable qui appartient au modèle de fusion ((*P1*) est vérifiée). Nous détaillons *(R2)* comme suit:

Obligatoire ○ Variable = {présent} U {présent, absent} = {présent, absent}= Variable. Par conséquent, la propriété *(P2)* est aussi vérifiée par *(R2)*.

Obligatoire ○ Absent = Variable (R3)

- *(R3)* spécifie que si un élément obligatoire appartenant à un des modèles en entrée ne correspond à aucun élément des autres modèles en entrée, alors l'élément résultant est un élément variable qui appartient au modèle de fusion ((*P1*) est vérifiée). Nous détaillons *(R3)* comme suit:

Obligatoire ○ Absent = {présent} U {absent} = {présent, absent} = Variable. Par conséquent, la propriété *(P2)* est aussi vérifiée par *(R3)*.

Variable ○ Variable = Variable (R4)

- *(R4)* spécifie que si un élément variable appartenant à un des modèles en entrée correspond à un élément variable qui appartient à un autre modèle en entrée, alors l'élément résultant de leur fusion est un élément variable qui appartient au modèle de fusion ((*P1*) est vérifiée). Nous détaillons *(R4)* comme suit:

Variable ○ Variable = {présent, absent} U {présent, absent} = {présent, absent}= Variable. Par conséquent, la propriété *(P2)* est aussi vérifiée par *(R4)*.

Variable ○ Absent = Variable (R1)

- *(R5)* spécifie que si un élément variable appartenant à un des modèles en entrée ne correspond à aucun élément des autres modèles en entrée, alors l'élément résultant est un élément variable qui appartient au modèle de fusion ((*P1*) est vérifiée). Nous détaillons *(R5)* comme suit:

Variable ○ Absent = {présent, absent} U {absent} = {présent, absent} = Variable. Par conséquent, la propriété *(P2)* est aussi vérifiée par *(R5)*.

Algorithme 1.a : Algorithme principal de fusion de modèles structurels

Entrée :

model1 : Model, *model2* : Model

Résultat :

Fusionner les modèles *model1* et *model2*

début

si *Match (model1, model2) != null* **alors**

| *resModel:* Model

| *resModel ← createModel()*

| *Algorithme1.b (model1, model2, resmodel)*

sinon

| *errorMessage()*

fin

La fusion des modèles structurels en entrée se base sur l'algorithme principal de fusion défini dans Algorithme1.a. Cet algorithme prend en entrée deux modèles structurels de lignes de produits, *model1* et *model2*, et réalise leur fusion. La fonction *Match* compare les deux signatures des modèles en entrée. Si les deux modèles correspondent, selon les définitions *(1) (2)* et *(3)*, alors le modèle, *resModel*, qui contiendra le résultat de fusion des deux modèles est créé et l'algorithme fait appel à un autre algorithme, Algorithme1.b, qui va réaliser la fusion des éléments contenus dans les deux modèles d'entrée. Dans le cas contraire, la fusion ne sera pas réalisée.

L'algorithme Algorithme1.b est défini afin de réaliser la fusion des éléments structurels contenus dans les modèles en entrée. Cet algorithme implémente les règles de fusion précédemment présentées et montre comment naviguer à travers les éléments des modèles en respectant leurs structures composites. Pour illustrer l'exécution de cet algorithme, nous utilisons l'exemple de la figure 4.8. Les modèles d'entrée sont les deux modèles de sécurité représentés dans la figure 4.8.(a) et la figure 4.8.(b). Ils sont fusionnés pour donner le modèle résultant de fusion illustré dans la figure 4.8.(c).

Algorithme 1.b : Algorithme de fusion des éléments structurels

Entrée :

elem1 : NamedElement, *elem2* : NamedElement, *res*: NamedElement

Résultat :

Fusionner les éléments *elem1* et *elem2*

```
1  début
2  //Création d'une liste de couples d'éléments, chaque couple inclut deux éléments
   correspondants
3  listMOfCouples ← MatchElem (elem1.ownedElements, elem2.ownedElements)
4  pour cp Є listMOfCouples faire
5  |  mergedChild : NamedElement
6  |  mergedChild ← CopyWithoutChildren(cp.get(1))
7  |  variabilityInfo (cp.get(1), cp.get(2), mergedChild )
8  |  res.ownedElements.add(mergedChild )
9  |  Algorithme1.b (cp.get(1), cp.get(2), mergedChild )
10 fin
11 //Création d'une liste des éléments qui n'ont pas de correspondants
12 listNotMatched ← NotMatchElem (elem1.ownedElements, elem2.ownedElements)
13 pour e Є listNotMatched faire
14 |  child : NamedElement
15 |  child ← CopyWithChildren(e)
16 |  variabilityInfo (e, null, child )
17 |  res.ownedElements.add(child )
18 fin
19 fin
```

L'algorithme Algorithme1.b prend deux modèles en entrée représentés par les variables *elem1* et *elem2*. Le résultat de leur fusion sera inclus dans *res* le modèle de fusion résultant. Il commence son traitement par la récupération des couples de classes correspondantes dans une liste *listMOfCouples.* Les éléments correspondants appartenant aux modèles d'entrée sont ainsi identifiés (ligne 3) par la fonction *MatchElem* pour qu'ils soient fusionnés par appel récursif de l'algorithme Algorithme1.b.

Dans notre exemple, les deux classes composites *Security* correspondent et sont ajoutées à la liste *listMOfCouples.* Pour les fusionner, une troisième classe composite *Security* est créée vide

(ligne 6), son information de variabilité est calculée en utilisant les règles de fusion précédemment décrites (ligne 7) (voir tableau 4.1) et par conséquent la classe *Security* créée porte l'information de variabilité Obligatoire.

La classe *Security* est ensuite insérée dans le modèle de fusion résultant (ligne 8). L'étape suivante consiste à fusionner récursivement les éléments contenus dans les classes composites *Security* qui appartiennent aux modèles d'entrée par appel récursif de l'algorithme Algorithme1.b. Le résultat de leur fusion est inséré dans la classe composite *Security* du modèle de fusion résultant. Cet appel récursif prend en entrée les deux classes composites *Security* à fusionner et le résultat de leur fusion sera inclus dans la classe *Security* précédemment créée. Les éléments correspondants appartenant aux classes composites d'entrée sont identifiés pour être fusionnés par appel récursif de l'algorithme Algorithme1.b. Dans l'exemple de la figure 4.8, le port *camControlSec* de la figure 4.8.(a) correspond au port *camControlSec* de la figure 4.7.(b) et sont alors fusionnés. D'autre part, les éléments qui sont contenus dans les classes composites *Security* des modèles d'entrée et qui ne correspondent à aucun autre élément sont récupérés dans la liste *listNotMatched*. Chaque élément de la liste *listNotMatched* est copié (ligne 15), son information de variabilité est calculée (ligne16) et il est enfin ajouté à la classe composite résultante *Security* (ligne 17). Nous citons à titre d'exemple le port *sirenSec* qui est contenu dans la classe composite *Security* de la figure 4.8.(b) et qui ne correspond à aucun port de la classe composite *Security* de la figure 4.8.(a). Par conséquent, il est copié, son information de variabilité est calculée, et il est finalement inséré dans la class composite *Security* du modèle de fusion résultant.

L'algorithme Algorithme1.b termine son exécution quand tous les éléments contenus dans les modèles d'entrée sont parcourus de manière récursive, fusionnés et quand le résultat de leur fusion est inséré dans le modèle de fusion résultant. Le modèle global de la ligne de produits *Security* illustré dans la figure 4.8.(c) ainsi obtenu représente le modèle qui résulte de la fusion des modèles d'entrée illustrés dans les figures 4.8.(a) et 4.8.(b).

Les règles de fusion implémentées dans l'algorithme Algorithme1.b permettent de réaliser la fusion des éléments structurels des modèles d'entrée. Le résultat de cette fusion est le modèle de fusion qui représente le modèle global de la ligne de produits. Ce modèle de fusion inclut tous les éléments structurels résultants ainsi que leurs informations de variabilité associées (voir les propriétés *(P1)* et *(P2)*). Cependant malgré la conservation de l'information de variabilité durant la fusion, le modèle de fusion résultant pourrait bien produire des systèmes qui ne sont valides pour aucun des modèles d'entrée. Pour restreindre les systèmes obtenus du

modèle de fusion tels que la propriété *(P3)* est respectée, nous proposons de combiner les contraintes de variabilité associées aux éléments structurels des modèles en entrée.

4.4.2.3. Fusion des contraintes de variabilité

Nous proposons un algorithme de fusion des contraintes de variabilité associées aux éléments structurels des modèles en entrée. L'algorithme Algorithme2 décrit comment les contraintes de variabilité sont combinées pour produire les contraintes associées aux éléments du modèle de fusion résultant, tel que la propriété *(P3)* est respectée. Le principe de l'algorithme consiste à faire évoluer l'ensemble des contraintes de variabilité de chaque modèle d'entrée pour ensuite réaliser leur disjonction. Comme exemple illustratif, nous poursuivons avec le cas d'étude des lignes de produits de sécurité présenté dans la figure 4.8.

Algorithme2 : Algorithme de fusion des contraintes de variabilité

```
Entrée :
// les éléments structurels des modèles d'entrée
E1 : list(NamedElement), E2 : list(NamedElement)
// les contraintes de variabilité des modèles d'entrée
C1 : list(Constraint), C2 : list(Constraint)
// les éléments variables du modèle de fusion résultant
E3 : list(NamedElement)
Résultat :
// les contraintes resultants
C3 : list(Constraint)
1   début
2   matchingElem ← Match (elem1, elem2)
3   C1' ← C1
4   C2' ← C2
5   pour e ∈ E3 faire
6     e1 ← findMatch (e, E1)
7     si e1 = null alors
8       const ← createConstraint (e, not)
9       add (const, C1')
10    fin
11    si e1 ! = null alors
12      si isMandatory (e1, E1) alors
13        const ← createConstraint (e, atLeastOne)
14        add (const, C1')
15      f in
16    fin
17    e2 ← findMatch (e, E2)
18    si e2 = null alors
19      const ← createConstraint (e, not)
20      add (const, C2')
21    fin
22    si e2 ! = null alors
23      si isMandatory (e2, E1) alors
24        const ← createConstraint (e, atLeastOne)
25        add (const, C2')
26      fin
27    fin
28  fin
29  C3 ← createConstraint ((C1', C2'), xor)
30  Return C3
31  fin
```

L'algorithme prend en entrée : *E1* l'ensemble des éléments structurels du premier modèle d'entrée, *E2* l'ensemble des éléments structurels du deuxième modèle d'entrée, *E3* l'ensemble des éléments variables contenus dans le modèle de fusion résultant suite à l'exécution de l'algorithme Algorithme1.a, *C1* l'ensemble des contraintes de variabilité associées aux éléments structurels du premier modèle d'entrée et *C2* l'ensemble des contraintes de variabilité associées aux éléments structurels du deuxième modèle d'entrée. *C1* et *C2* sont copiés respectivement dans *C1'* et *C2'* pour les faire évoluer durant l'exécution de l'algorithme. Les éléments de *E3* sont ensuite parcourus et comparés avec les éléments de *E1* et de *E2*. Dans l'exemple de la figure 4.8, la part *securityManager* contenue dans le modèle de fusion (figure 4.8.(c)) ne correspond à aucun élément du premier modèle (figure 4.8.(a)), par conséquent la part *securityManager* ne doit être présente dans aucun des produits valides calculés à partir de *C1'*. Pour cela, une contrainte de type *not* est associée à *securityManager* et ajoutée à *C1'* (ligne 5 à ligne 8). D'autre part, la part *securityManager* contenue dans le modèle de fusion (figure 4.8.(c)) correspond à la part obligatoire *securityManager* contenue dans le deuxième modèle (figure 4.8.(b)). Une contrainte de type *atLeastOne* est alors associée à la part *securityManager* pour indiquer qu'elle doit être présente dans tous les produits valides calculés à partir de *C2'* (ligne 22 à ligne 25). Une fois que tous les éléments d'*E3* sont parcourus et traités, l'algorithme crée une contrainte *C3* de type xor entre *C1'* et *C2'*, et termine ainsi son traitement. Ceci implique pour le cas de la part *securityManager* qu'elle ne peut être présente dans aucun des produits valides du premier modèle. Elle est, par contre, présente dans tous les produits valides du deuxième modèle. Ainsi, la propriété *(P3)* est respectée.

La complexité (au pire cas) des trois algorithmes (fusion d'éléments structurels et fusion des contraintes de variabilité) présentés dans cette section est polynomiale (O (n^p)) avec $p > 0$ et n représente le nombre d'éléments structurels à fusionner.
Le nombre maximal de n est atteint dans le cas où pour trouver le correspondant de chaque élément et les fusionner, il faut parcourir tous les éléments. Le tableau 4.2 présente les caractéristiques des trois algorithmes, précédemment présentés, en termes de complexité et de terminaison (algorithme ne bouclant pas infiniment).

	Algorithme1.b	**Algorithme1.a**	**Algorithme2**
Calcul de complexité	3 niveaux de profondeur dans les modèles : -**Niveau1** : niveau modèle : calcul de la liste des classes correspondantes → $O(n^2)$ + calcul de la liste des classes qui ne correspondent pas → $O(n^2)$ - **Niveau2** : niveau classe : Une boucle de n itérations dont chacune contient : calcul de la liste des propriétés correspondantes → $O(n^2)$ + calcul de la liste des propriétés qui ne correspondent pas → $O(n^2)$ **Niveau3** : niveau propriétés (ports et parts) : Une boucle de n itérations dont chacune contient : $O(1)+O(1)$ car les propriétés ne contiennent pas d'autres éléments. Ceci peut être exprimé comme suite : $O(n^2) + O(n^2) + n\ (O(n^2)+ O(n^2)+ n(O(1)+ O(1)))$ **Complexité totale → $O(n^3)$**	Complexité de l'Algorithme1.b **Complexité totale → $O(n^3)$**	Une boucle de n itérations contenant l'appel d'une autre boucle (dans *findMatch*) de n itérations. Donc deux boucles imbriquées. **Complexité totale → $O(n^2)$**
Absence de boucles infinies	Présence de 3 niveaux dans le modèle où chaque niveau contient n éléments dont le nombre est connu à l'avance.	Pareil qu'Algorithme1.b, présence de 3 niveaux dans le modèle où chaque niveau contient n éléments dont le nombre est connu à l'avance.	Présence de 2 boucles dont le nombre d'itérations est connu à l'avance.

Tableau 4.2– Les caractéristiques des 2 algorithmes de fusion en termes de complexité et de terminaison

Il ne s'agit pas d'une preuve par récurrence pour la terminaison ni d'un calcul détaillé de la complexité. Toutefois, il est important de considérer ces caractéristiques et de s'assurer que ces algorithmes répondent bien au but de l'utilisateur qui est la fusion de modèles de lignes de produits.

4.4.3. Synthèse

Nous avons vu dans cette section que la fusion des modèles de lignes de produits logiciels, représentés par des structures composites d'UML, inclut la fusion des éléments structurels ainsi que la fusion des contraintes de variabilité. Pour donner un sens à la fusion, nous avons défini un ensemble de propriétés sémantiques qui doivent être satisfaites au cours de la fusion. C'est en respectant ces propriétés que nous avons proposé un catalogue de règles pour la fusion des éléments structurels ainsi qu'un algorithme de fusion des contraintes de variabilité. Pour la comparaison des éléments structurels à fusionner, nous avons défini des critères génériques applicables sur des éléments UML. Nous avons ensuite spécialisé ces critères pour définir d'autres critères de correspondance spécifiques aux concepts manipulés. Les règles de fusion proposées permettent de calculer l'information de variabilité associée à chacun des éléments structurels du modèle de fusion résultant. Nous avons aussi proposé des algorithmes qui implémentent ces règles en respectant la structure composite des lignes de produits fusionnées. Pour la fusion des contraintes de variabilité, nous avons proposé un algorithme générique qui ne dépend pas des types de contraintes manipulées. L'utilisation d'un tel algorithme permet au concepteur de réaliser la fusion des contraintes de variabilité quelques soient leurs types, y compris les contraintes transversales de types *implication* et *équivalence*.

4.5. Consolidation du modèle de fusion résultant : troisième phase de fusion

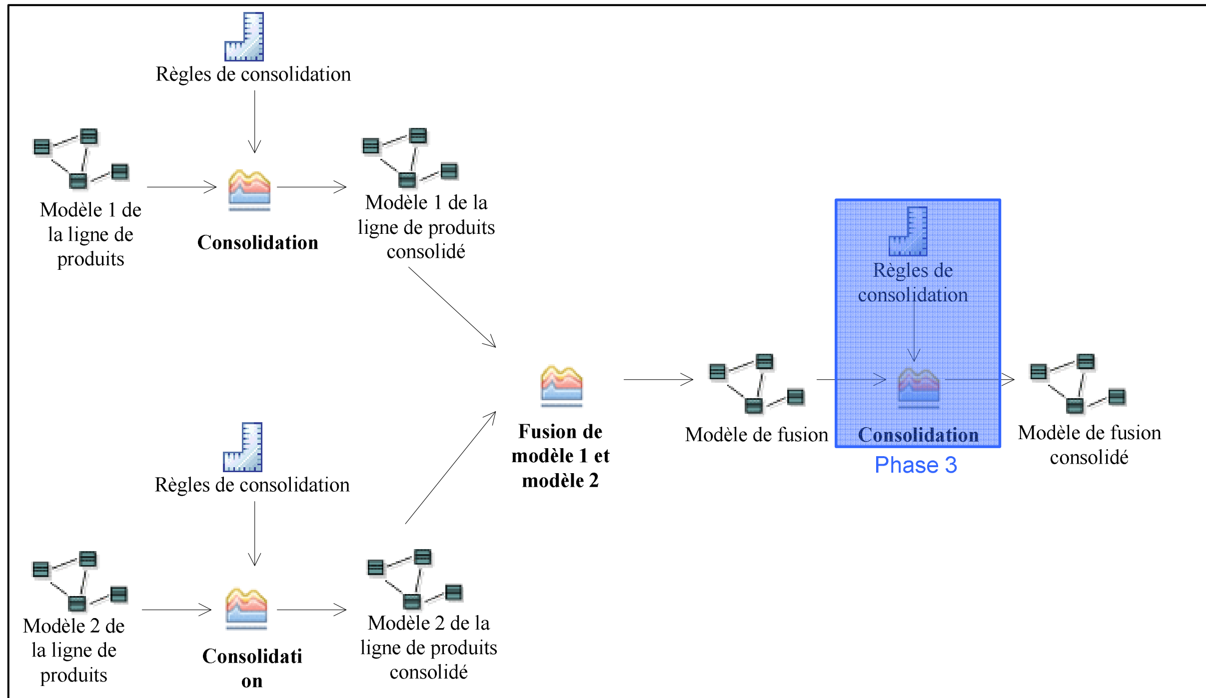


Figure 4.9: La phase de consolidation post fusion

La phase de consolidation du modèle de fusion résultant représente la dernière phase de la démarche globale de fusion, voir figure 4.9.

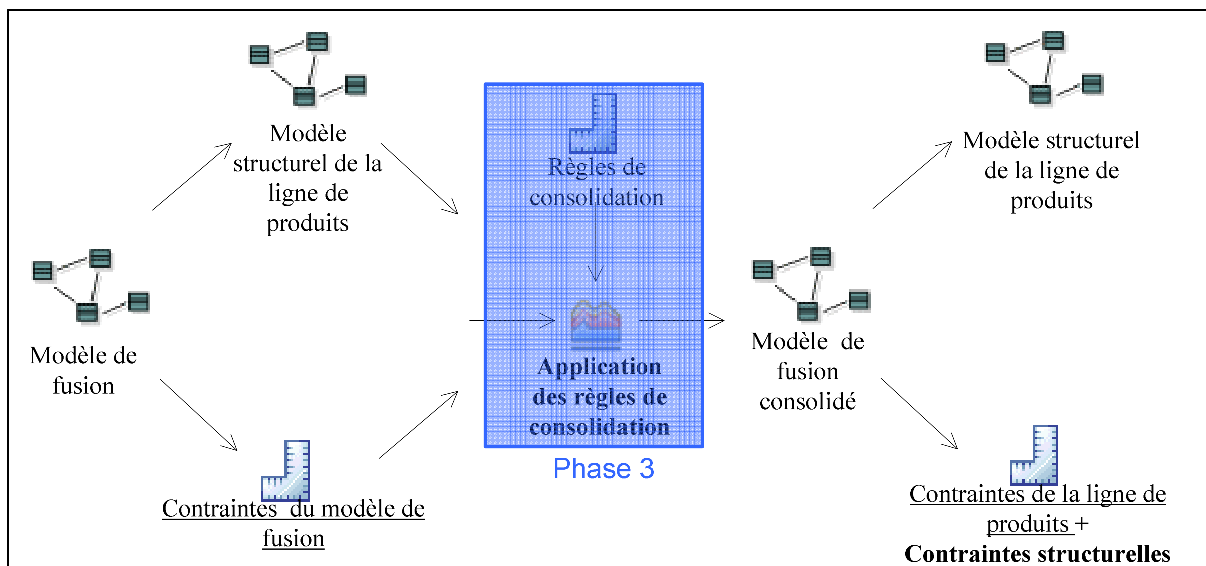


Figure 4.10 : Vue détaillée de la phase de consolidation post fusion

Cette phase consiste à consolider le modèle de fusion résultant. Les règles de consolidation présentées dans le chapitre précédent sont appliquées sur le modèle de fusion résultant. Nous

rappelons que ces mêmes règles ont été appliquées sur les modèles en entrée. La ré-application de ces règles après la fusion s'explique par la présence d'éléments dont l'information de variabilité est devenue variable dans le modèle de fusion résultant. La figure 4.10 montre que l'application des règles de consolidation sur le modèle de fusion résultant se matérialise par l'ajout de contraintes structurelles. Ces contraintes permettent l'obtention de modèles de produits structurellement cohérents et complets.

Nous continuons avec l'exemple du modèle de fusion de la ligne de produits de sécurité, *Security*, obtenu à l'issu de la phase de fusion et illustré dans la figure 4.8. Nous remarquons que les nouveaux éléments variables dans ce modèle sont les suivants : *securityManager*, *secInstruct*, *secInstructSec*, *detectInstruct* et *detectInstructSec*. Il s'agit des éléments annotés comme variables dans le modèle de fusion et qui n'ont été variables dans aucun modèles d'entrée. Par exemple, la part *securityManager* est une part variable contenue dans le modèle de fusion. Elle provient de la part fixe *securityManager* du premier modèle d'entrée comme nous pouvons le percevoir dans la figure 4.8.(a). L'application des règles de consolidation sur le modèle de fusion a comme résultat le modèle de fusion consolidé avec l'ensemble des contraintes structurelles suivantes :

- Par application de *(Reg1)* les contraintes ajoutées sont :

C_a = *Equivalence (secInstruct, secInstructSec)*

C_b = *Equivalence (detectInstruct, detectInstructSec)*

Par application de *(Reg3)* les contraintes ajoutées sont :

C_c = *Implication (secInstruct, securityManager)*

C_d = *Implication (alarmController, securityManager)*

C_e = *Implication (detectController, securityManager)*

C_f = *Implication (detectInstruct, detector)*

- Par application de *(Reg4)* les contraintes ajoutées sont :

C_g = *Implication (alarmInstruct, securityManager)*

C_h = *Implication (alarmEn, securityManager)*

C_i = *Implication (detectI, securityManager)*

C_j = *Implication (detectEn, securityManager)*

4.6. Conclusion

Le mécanisme de fusion permet de combiner des modèles de lignes de produits ayant des parties communes. Nous avons détaillé les phases du processus de fusion proposé. La première phase a pour objectif de consolider les modèles de lignes de produits en entrée en leurs appliquant un ensemble de règles. Ces règles permettent d'obtenir des modèles d'entrée consistants et prêts à être fusionnés. La deuxième phase réalise la fusion des modèles d'entrée. Cette phase est constituée de deux étapes principales dont la première s'intéresse à identifier les critères de correspondance entre les éléments structurels manipulés alors que la seconde réalise la fusion des éléments identifiés comme correspondants. La seconde étape spécifie la sémantique de la fusion proposée. Elle s'intéresse à la fusion des éléments structurels en s'appuyant sur un ensemble de règles de fusion. Un algorithme de fusion est présenté pour implémenter ces règles de fusion en respectant la structure composite des éléments ainsi que l'information de variabilité associée à chacun d'entre eux. Cette étape réalise aussi la fusion des contraintes de variabilité associées aux éléments des modèles en entrée. Le modèle de fusion résultant est ainsi caractérisé par des éléments structurels dont l'information de variabilité a été calculée et modifiée ainsi que par l'ensemble des contraintes combinées. La troisième phase du processus a pour but de consolider le modèle de fusion résultant, vu que de nouveaux éléments sont devenus variables durant l'étape de fusion. Le modèle de fusion résultant est ainsi consolidé et permet d'obtenir des modèles de produits structurellement cohérents et complets. Dans le chapitre suivant, nous détaillerons la démarche proposée pour l'agrégation des modèles de lignes de produits logiciels.

CHAPITRE 5

Agrégation des modèles de lignes de produits logiciels

5.1. Exemple illustratif 87
5.2. Démarche d'agrégation : aperçu global 88
5.3. Consolidation des modèles en entrée : première phase d'agrégation 90
5.4. Agrégation des modèles en entrée : deuxième phase d'agrégation 93
5.4.1. Propriété sémantique de l'agrégation 96
5.4.2. Proposition1 97
5.4.2.1. Règles d'agrégation 97
5.4.2.2. Synthèse 103
5.4.3. Proposition2 103
5.4.3.1 Règles d'agrégation : 103
5.4.3.2. Synthèse 109
5.4.4. Agrégation et effets de bord 110
5.5. Conclusion 111

Ce chapitre a pour objectif de présenter la démarche proposée pour l'agrégation des modèles de lignes de produits logiciels. Il est structuré en quatre sections. La première section présente un exemple illustratif qui sera utilisé tout au long de la démarche d'agrégation. La deuxième section donne un aperçu global de la démarche. La troisième section présente la première phase responsable de la consolidation des modèles à agréger. Enfin, la quatrième section détaille les traitements effectués pour l'agrégation des modèles d'entrée consolidés. Dans cette dernière section deux propositions sont présentées pour réaliser l'agrégation. Elles sont suivies d'une estimation de certains effets de bord susceptibles d'apparaitre comme conséquences de l'agrégation.

5.1. Exemple illustratif

Nous poursuivons avec l'exemple d'agrégation présenté dans le chapitre 3 dont nous reprenons les modèles dans les figures ci-dessous. Il s'agit du modèle de la ligne de produits de sécurité « Détecteur de présence avec alarme» et du modèle de la ligne de produits de caméra surveillance:

- Modèle de ligne de produits de sécurité « Détecteur de présence avec alarme » : la figure 5.1 présente le modèle de ligne de produits de sécurité décrit dans le chapitre 3 (voir la page 40).

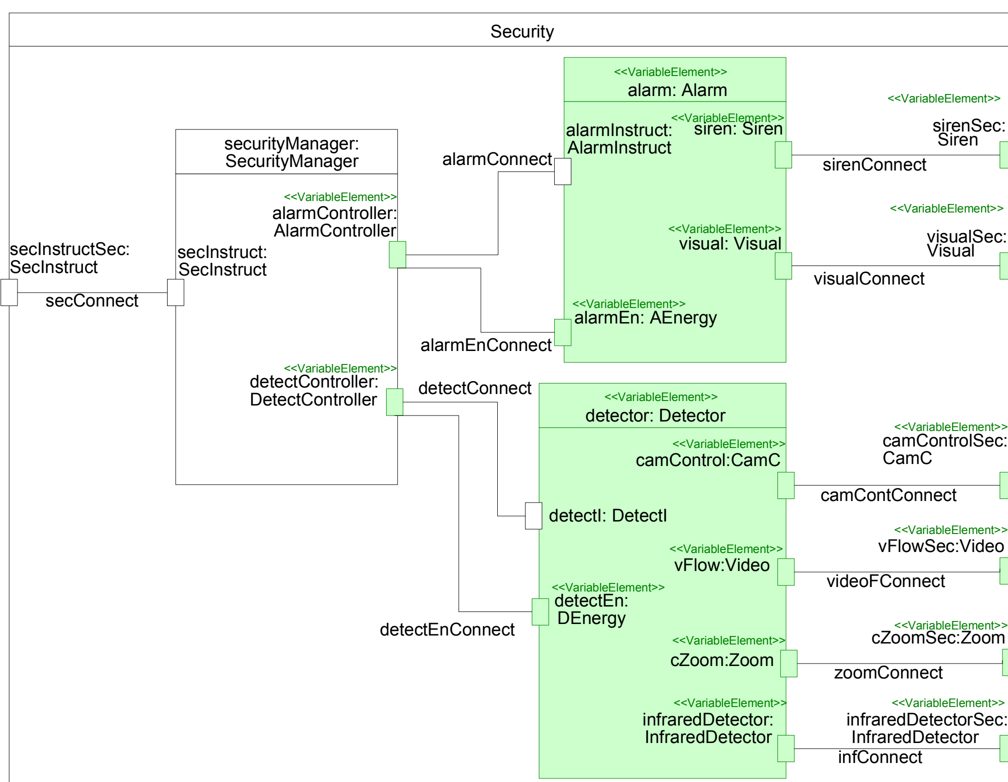


Figure 5.1: Modèle de la ligne de produits de sécurité « Détecteur de présence avec alarme»

- Modèle de ligne de produits de caméra surveillance : La figure 5.2 présente le modèle de la ligne de produits de caméra surveillance. La ligne *Camera* est gérée par la part fixe *camManager*. Le contrôle de l'énergie est réalisé via les ports variables *camEnC* et *camEn* ainsi que les ports fixes *camEnergy* et *camEnergyC*. Le focus et la position de la caméra sont gérés par les ports respectifs *focus*, *focusC* et *positionControl*, *positionControlC*. Les enregistrements vidéo sont transmis à travers les ports *videoFlow* et *videoFlowC*. La ligne de produits possède une option de zoom représentée par la part variable *zoomC*. Cette option peut être gérée par la part *camManager* à travers les ports *zController* et *zInst*. Elle peut être aussi gérée via les ports *zoomInstruct* et *zoomInstructC*. La classe composite *Camera* offre ses services de surveillance via le port *camInstructC*.

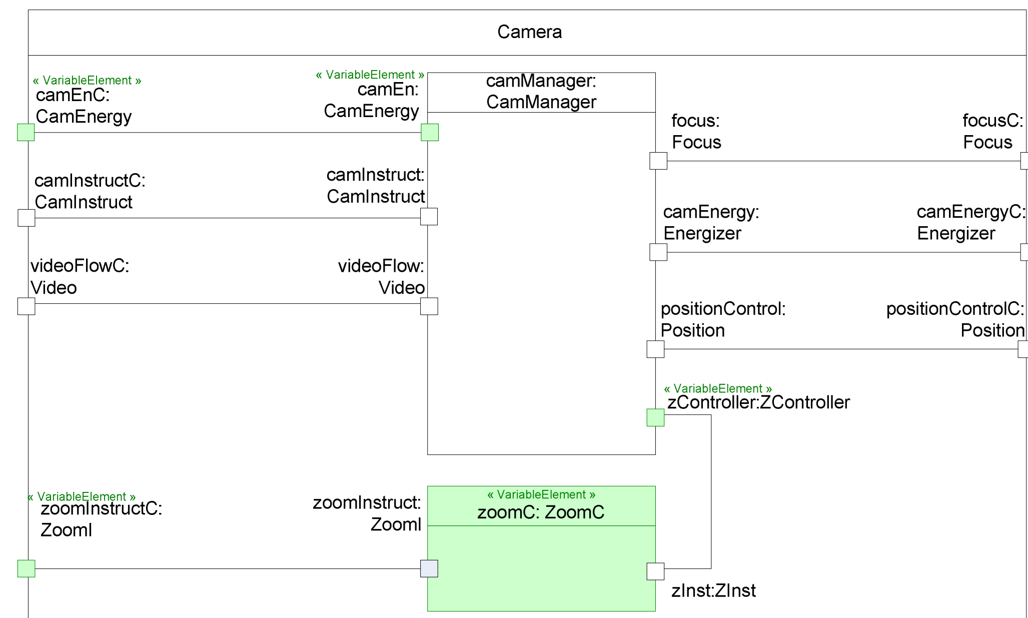


Figure 5.2 : Modèle de la ligne de produits de caméra surveillance

Comme indiqué sur la figure 3.16 du chapitre 3, les deux modèles sont reliés par une contrainte transversale associée aux éléments variables *detectEn* et *camEn* appartenant respectivement à *Security* et *Camera*.

Les deux modèles présentés ci-dessus nous serviront d'exemple illustratif pour la description de la démarche d'agrégation proposée dans la suite.

5.2. Démarche d'agrégation : aperçu global

La figure 5.3 donne un aperçu global sur la démarche d'agrégation. En effet, nous distinguons deux principales phases:

- La première phase, appelée phase de consolidation, a pour objectif d'assurer que les modèles réutilisés dans l'optique d'agrégation soient structurellement cohérents et complets. Nous utilisons pour cet objectif un ensemble de règles de consolidation, qui une fois appliquées sur les modèles en entrée, permettent d'obtenir des modèles consolidés et prêts pour l'agrégation.
- La seconde phase, nommée phase d'agrégation, inclut les différents traitements effectués pour l'agrégation des modèles de lignes de produits. A l'issue de cette phase, nous obtenons un modèle d'agrégation.

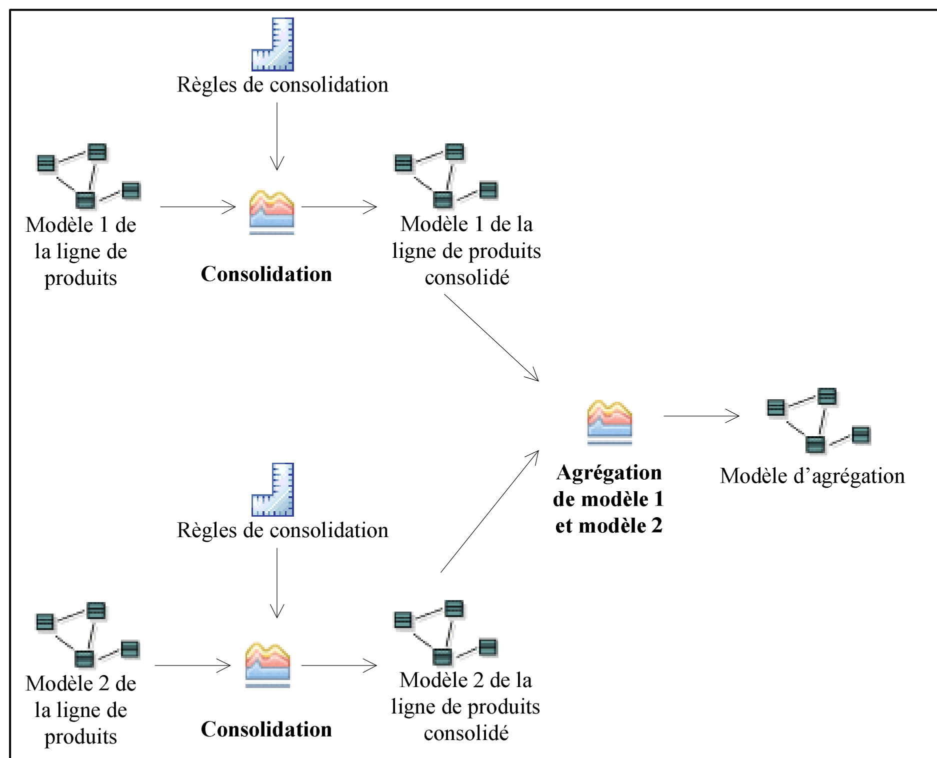


Figure 5.3 : Vue globale de la démarche d'agrégation

Suite à la description de la vue globale de la démarche d'agrégation, nous détaillerons les deux phases qui la constituent dans la suite.

5.3. Consolidation des modèles en entrée : première phase d'agrégation

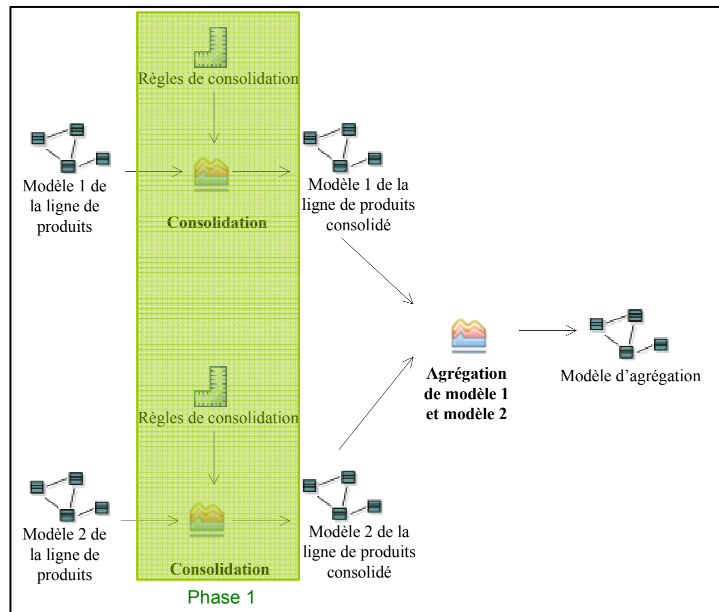


Figure 5.4 : La phase de consolidation

La phase de consolidation des modèles en entrée représente la première phase de la démarche globale d'agrégation, voir figure 5.4.

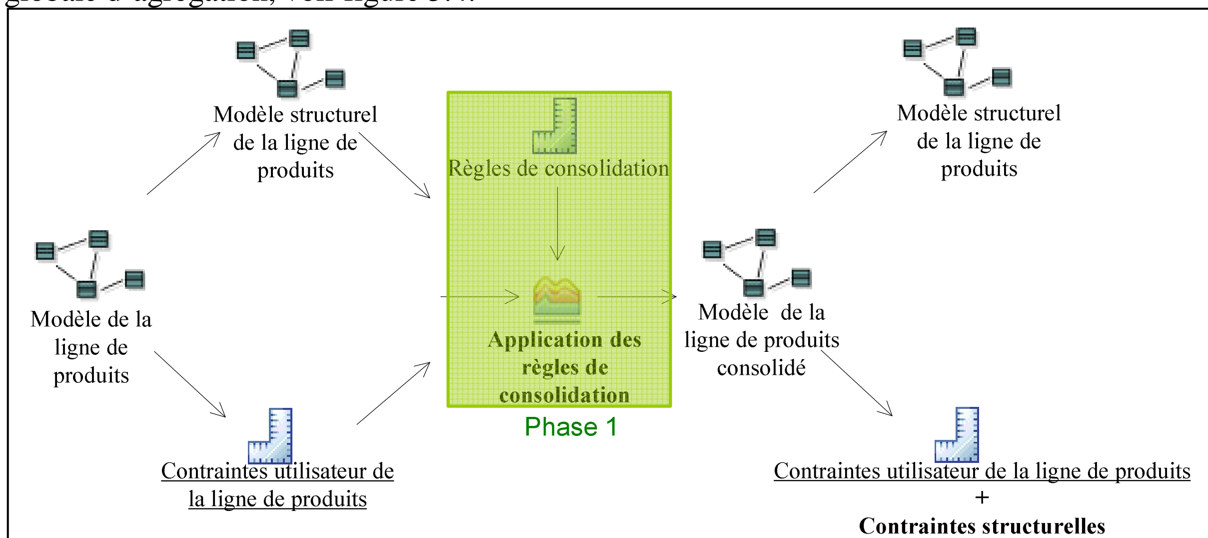


Figure 5.5 : Vue détaillée de la phase de consolidation

Cette phase consiste à consolider les modèles en entrée pour l'agrégation. Pour chacun des modèles de lignes de produits, la phase de consolidation consiste à appliquer un ensemble de règles définies dans la section 3.2 du chapitre 3. Ces règles ont pour conséquences la modification de l'information de variabilité de certains éléments structurels ainsi que par l'ajout d'un nouvel ensemble de contraintes structurelles (voir figure 5.5). Le but de ces règles est de guider le concepteur dans la construction d'un modèle de ligne de produits complet en

termes de contraintes structurelles lui permettant d'obtenir des modèles de produits structurellement cohérents et complets.

L'application des règles de consolidation sur les exemples du modèle de ligne de produits de sécurité, illustré dans les figures 5.1, et du modèle de ligne de produits de caméra surveillance, présenté dans la figure 5.2, donne les résultats suivants :

- Pour le modèle de ligne de produits de sécurité « Détecteur de présence avec alarme», l'application des règles de consolidation permet d'obtenir le modèle consolidé présenté dans la figure 5.6 avec l'ensemble des contraintes structurelles suivantes :

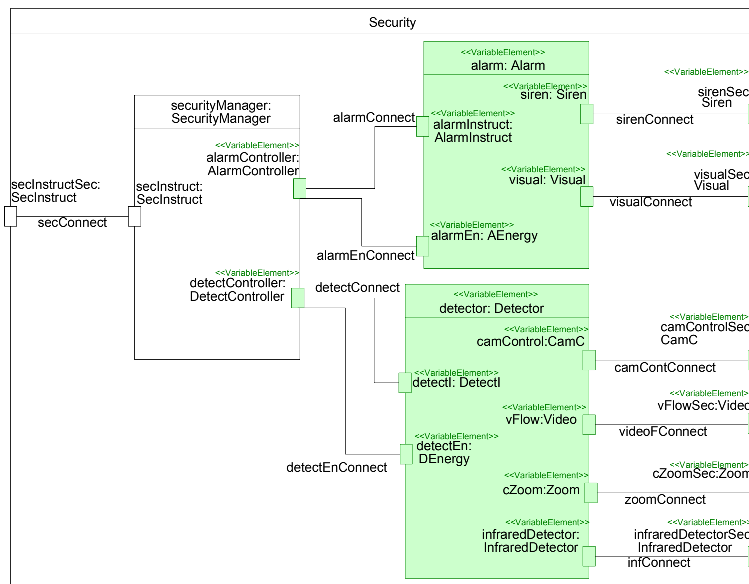


Figure 5.6 : Modèle de la ligne de produits Security après consolidation

Par application de *(Reg1)* les contraintes ajoutées sont :

C_5= *Equivalence (siren, sirenSec)*

C_{10} = *Equivalence (visual, visualSec)*

C_{11} = *Equivalence (camControl, camControlSec)*

C_{12} = *Equivalence (vFlow, vFlowSec)*

C_{13} = *Equivalence (cZoom, cZoomSec)*

C_{14} = *Equivalence (infraredDetector, infraredDetectorSec)*

Par application de *(Reg2)* les contraintes ajoutées sont :

C_6 = *Implication (alarmInstruct, alarmController)*

C_7 = *Implication (alarmEn, alarmController)*

C_{15} = *Implication (detectI, alarmController)*

C_{16} = *Implication (detectEn, alarmController)*

Par application de *(Reg3)* les contraintes ajoutées sont :

C_8= *Implication (siren, alarm)*

C_{17} = *Implication (visual, alarm)*

C_{18} = *Implication (alarmInstruct, alarm)*

C_{19} = *Implication (alarmEn, alarm)*

C_{20} = *Implication (camControl, detector)*

C_{21} = *Implication (vFlow, detector)*

C_{22} = *Implication (cZoom, detector)*

C_{23} = *Implication (infraredDetector, detector)*

C_{24} = *Implication (detectI, detector)*

C_{25} = *Implication (detectEn, detector)*

Par application de *(Reg4)* les contraintes ajoutées sont :

C_9 = *Implication (alarmController, alarm)*

C_{26} = *Implication (detectController, detector)*

- Pour le modèle de ligne de produits de caméra surveillance, l'application des règles de consolidation permet d'obtenir le modèle consolidé présenté dans la figure 5.7 avec l'ensemble des contraintes structurelles suivantes :

Par application de *Reg1* les contraintes ajoutées sont :

C_A= *Equivalence (zoomInstruct, zoomInstructC)*

C_B= *Equivalence (camEn, camEnC)*

C_C= *Equivalence (zController, zInst)*

Par application de *Reg3* les contraintes ajoutées sont :

C_D= *Implication (zoomInstruct, zoomC)*

C_E= *Implication (zInst, zoomC)*

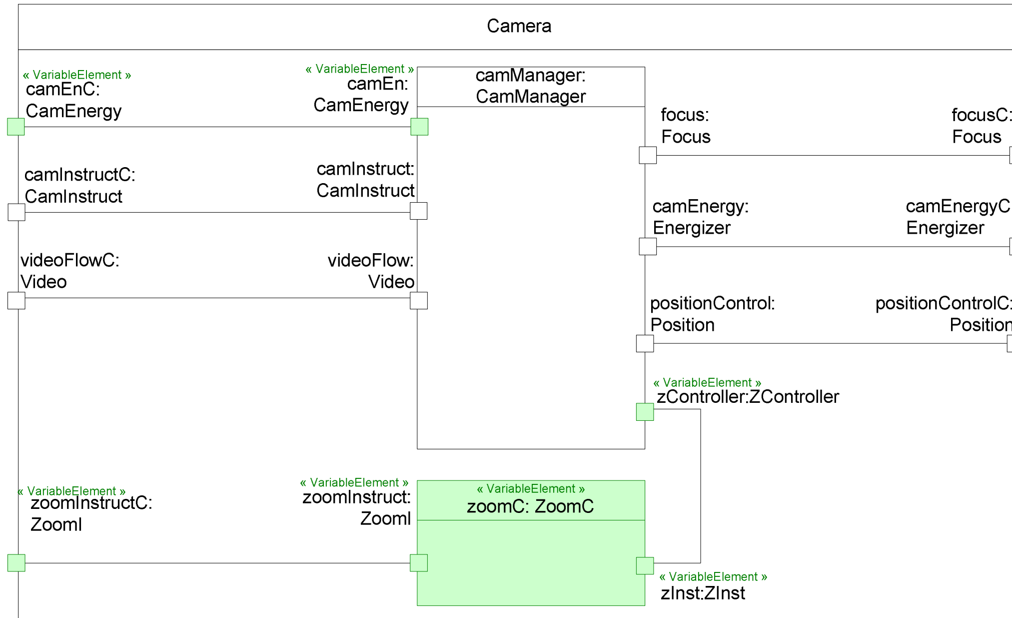


Figure 5.7 : Modèle de la ligne de produits de caméra surveillance après consolidation

5.4. Agrégation des modèles en entrée : deuxième phase d'agrégation

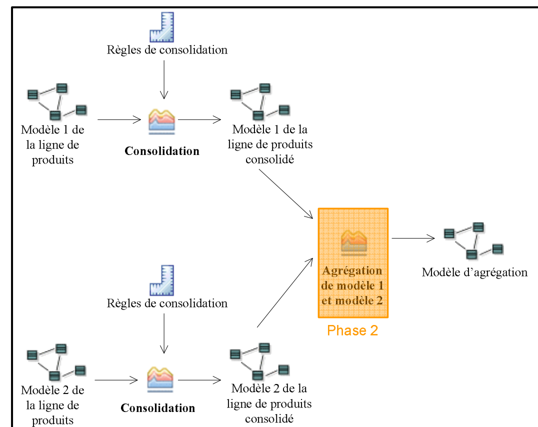


Figure 5.8: La phase d'agrégation

La phase d'agrégation des modèles en entrée représente la deuxième phase de la démarche globale d'agrégation (voir figure 5.8).

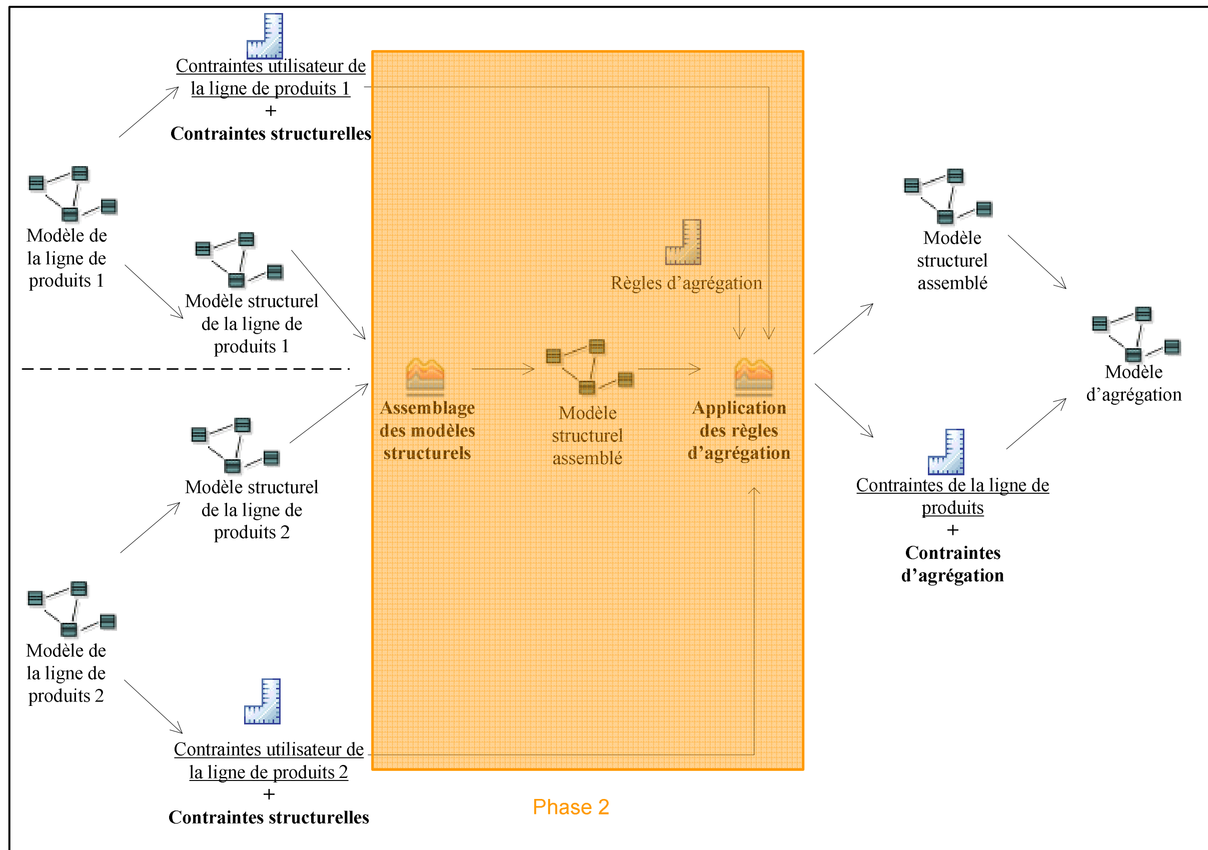


Figure 5.9 : Vue détaillée de la phase d'agrégation

L'objectif de cette phase est de combiner deux modèles de lignes de produits, qui ne présentent pas de similarités, en gérant la variabilité de leurs éléments d'interface connectés.

La figure 5.9 montre que cette phase est composée de deux étapes dont la première étape consiste à assembler les modèles structurels en entrée. Cet assemblage est réalisé par le concepteur qui doit intervenir pour assurer la connexion des éléments d'interface des modèles. Par exemple, il doit identifier les ports externes à connecter ainsi que leurs modes de connexion. Nous ne sommes pas chargés d'intervenir à ce niveau.

Nous intervenons dans le cadre de la deuxième étape à l'issue de l'étape d'assemblage afin de gérer les éléments variables identifiés pour la connexion. Nous leurs appliquons un ensemble de règles que nous appelons règles d'agrégation. Ces règles ont pour objectif de contraindre les éléments externes impliqués dans la connexion, c'est-à-dire ports et connecteurs d'assemblage, afin de maintenir les deux modèles de lignes de produits connectés tels que les connexions précédemment identifiées par le concepteur soient présentes dans les modèles de produits obtenus à partir le la ligne de produits constituées des deux modèles agrégés. A l'issue de cette phase, nous obtenons un modèle d'agrégation qui possède deux vues

complémentaires ; un modèle structurel assemblé et un ensemble de contraintes étendu par l'ajout des contraintes d'agrégation.

Vu que le concepteur est le seul responsable de la réalisation de l'étape d'assemblage et vu que nous n'intervenons que pour l'étape d'agrégation qui la suit, nous considérons que l'assemblage des modèles doit respecter certaines règles dont les hypothèses sont les suivantes :

L'assemblage des classes composites doit être possible et les ports à connecter doivent être bien compatibles. Nous ne nous intéressons qu'à la gestion de la variabilité des éléments de connexion pour maintenir les deux classes composites bien assemblées. Nous supposons donc que :

- Le connecteur inclut deux rôles : un attaché aux ports ayant le type d'une interface fournie et un autre attaché aux ports ayant le type d'interface requise.
- Deux ports peuvent être connectés si et seulement s'ils ont des interfaces compatibles et si un des ports est « requis » et l'autre est « fourni ».
- Un connecteur d'assemblage doit uniquement être défini à partir d'une interface requise ou un port vers une interface fournie ou un port.
- L'ensemble des services fournis par le port ou l'interface doit être un sur-ensemble de l'ensemble des services requis.

La figure 5.10 montre l'exemple d'assemblage des deux classes composites *Security* et *Camera*.

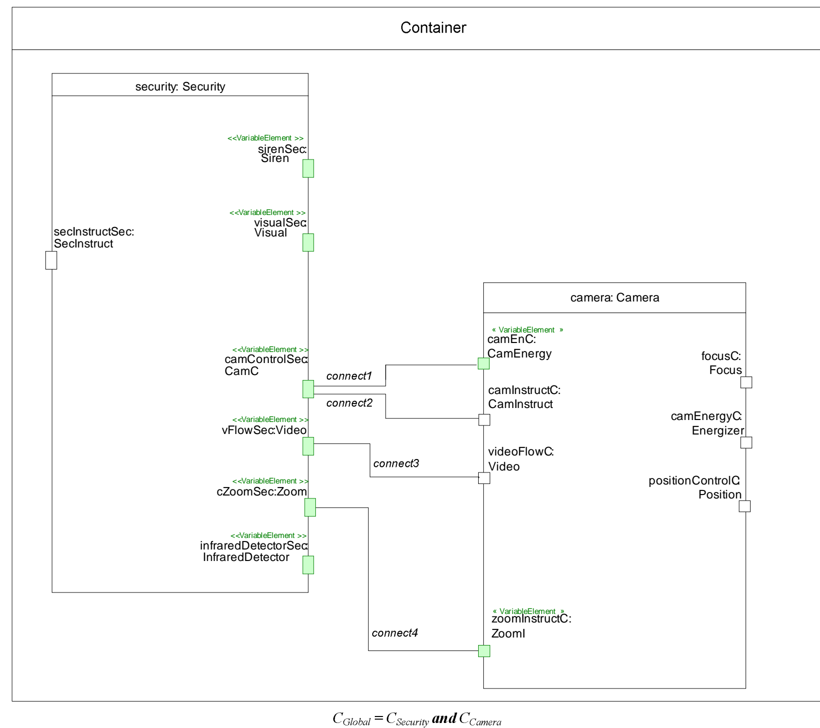


Figure 5.10 : Assemblage des classes composites Security et Camera

Les connexions identifiées par l'utilisateur pendant l'étape d'assemblage sont :

Le connecteur *connect1* établit la connexion entre les ports *camControlSec* et *camEnC*.

Le connecteur *connect2* établit la connexion entre les ports *camControlSec* et *camInstrucC*.

Le connecteur *connect3* établit la connexion entre les ports *vFlowSec* et *videoFlowC*.

Le connecteur *connect4* établit la connexion entre les ports *cZoomSec* et *zooInstructC*.

D'autre part, nous décrivons les contraintes de la ligne de produits assemblée $C_{assemblage}$ par la conjonction des contraintes, $C_{Security}$, du modèle consolidé de la ligne de produits de *Security* et les contraintes, C_{Camera}, du modèle consolidé de la ligne de produits *Camera*.

5.4.1. Propriété sémantique de l'agrégation

La définition des propriétés sémantiques de l'agrégation se base sur la relation entre l'ensemble des systèmes valides obtenus à partir du modèle d'agrégation résultant et l'ensemble des systèmes valides obtenus à partir des modèles en entrée.

Plus précisément, le résultat de l'agrégation doit satisfaire la propriété suivante :

Un système valide obtenu du modèle d'agrégation résultant représente l'assemblage de systèmes valides dont chacun est obtenu à partir de l'un des modèles en entrée. *(P_{Ag})*

Nous présentons dans la suite deux propositions pour réaliser l'agrégation des modèles de lignes de produits tels que la propriété *(P_{Ag})* est satisfaite.

5.4.2. Proposition1

La première alternative suppose que toutes les connexions identifiées entre les éléments externes des modèles de lignes de produits en entrée sont obligatoires. Elles doivent être établies automatiquement. Pour ce faire, la proposition1 s'intéresse à gérer les ports variables des classes composites représentant les modèles d'entrée en se basant sur un ensemble de règles d'agrégation.

5.4.2.1. Règles d'agrégation

***Règle 1* :** Si deux ports sont connectables (hypothèse) et qu'ils doivent être connectés, et si l'un des deux ports est variable alors une contrainte doit être créée pour assurer que le port variable soit présent pour établir la connexion.

C'est par exemple le cas du port variable *vFlowSec* attaché à la classe composite *Security* et qui est connecté au port obligatoire *videoFlowC* attaché à la classe composite *Camera*.

***Règle 2* :** Si deux ports simples sont connectables (hypothèse) et qu'ils doivent être connectés, et si les deux ports sont variables alors une contrainte doit être créée pour assurer que chacun des ports soit présent pour établir la connexion.

Par exemple le port variable *cZoomSec* attaché sur la classe composite *Security* est connecté au port variable *zoomInstructC* attaché sur la classe composite *Camera*.

***Règle 3* :** Si deux ports variables sont connectables (hypothèse) et qu'ils doivent être connectés, et si au moins un des deux ports est multiple alors une contrainte doit être créée pour assurer que la présence du port non multiple (ou multiple si les deux ports sont multiples) dans tous les systèmes valides implique celle du port multiple.

Prenons l'exemple du port variable *camEnergy* attaché sur la classe composite *Camera* et qui est connecté au port variable multiple *camControlSec* attaché sur la classe composite *Security*.

Algorithme d'agrégation **:**

Nous proposons un algorithme d'agrégation des modèles de lignes de produits en entrée représentés sous une vue de structure composite. L'algorithme AlgorithmeP1 implémente les règles d'agrégation précédemment décrites pour la proposition1 et montre comment les contraintes d'agrégation sont ajoutées afin d'assurer les connexions entre les ports externes des classes composites telles que la propriété (P_{Ag}) est respectée. Pour illustrer l'exécution de cet algorithme, nous continuons avec l'exemple de la figure 5.10.

L'algorithme AlgorithmeP1 prend en entrée la liste des connexions *Connexions* obtenue à l'issue de l'étape d'assemblage. Chaque connexion est définie par les deux ports connectés ainsi que par le lien de connexion choisi pour l'établir, qui peut être un connecteur par exemple. *C1* est l'ensemble des contraintes du premier modèle de lignes de produits modèle1. *C2* est l'ensemble des contraintes du deuxième modèle de lignes de produits modèle2. *Ctrans* est l'ensemble des contraintes transversales. *C1*, *C2* et *Ctrans* sont aussi considérés en entrée de l'algorithme d'agrégation. Le résultat est l'ensemble final des contraintes *Cres* obtenu suite à l'application de l'algorithme.

Le traitement commence par l'ajout des listes de contraintes *C1*, *C2* et *Ctrans* à l'ensemble de contraintes résultant *Cres*. L'algorithme va ensuite parcourir la liste de connexions *Connexions* pour appliquer les règles d'agrégation à chacune d'entre elles. Nous identifions deux principales parties de traitement.
La première partie (ligne 7 – ligne17) implémente la première règle d'agrégation où un seul des ports d'une connexion donnée est variable. Donc pour une connexion identifiée, si le premier port est variable (ligne 7) et le deuxième port est fixe (ligne 8) alors une contrainte de type *atleastOne* est ajoutée (lignes 9 et 10) pour assurer la présence du port variable afin de réaliser cette connexion. Idem, si le premier port est fixe (ligne 12) et le deuxième port est variable (ligne 13) alors une contrainte de type *atleastOne* est ajoutée (lignes 14 et 15) pour assurer la présence du port variable afin de réaliser cette connexion.
La deuxième partie (ligne 18 – ligne 42) implémente les règles 2 et 3 où les deux ports d'une connexion donnée sont variables. Dans le cas où les deux ports sont simples (lignes 20 et 21), c'est-à-dire ils ne participent qu'à une seule connexion, deux contraintes de types respectifs *equivalence* et *atLeastOne* sont ajoutées et spécifient que pour assurer la connexion à laquelle ils participent, les deux ports doivent être présents ensemble (lignes 22 à 24). Dans le cas où au moins l'un des deux ports est multiple (ligne 27 et ligne 33), alors deux contraintes une de

type *implication* et l'autre de type *atLeastOne* sont ajoutées. Elles spécifient que le choix du premier port, qui doit être toujours présent, implique le choix du port multiple (lignes 28 à 31) et (lignes 34 et 37) et que ces ports variables doivent être présents pour les connexions. Une fois que toutes les connexions définies par les ports connectés et les connecteurs qui les relient sont parcourues, l'algorithme termine son traitement et retourne l'ensemble des contraintes résultantes.

AlgorithmeP1: algorithme d'agrégation de la proposition 1

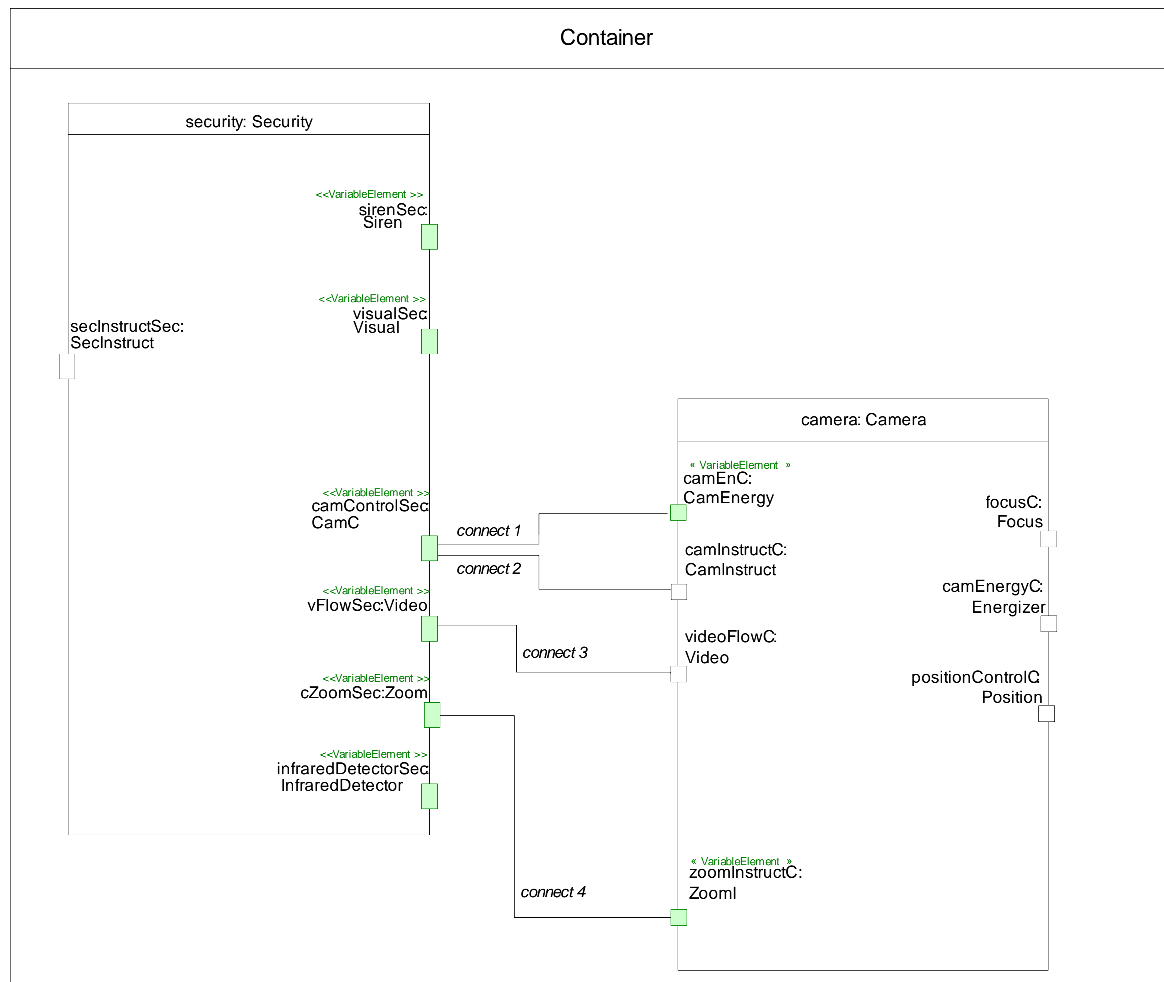
```
Entrée :
Connexions:      //la liste de triplets  ports connectés et connecteur
C1 :             // la liste de contraintes du modèle 1 de la ligne de produits.
C2 :             // la liste de contraintes du modèle 2 de la ligne de produits.
Ctrans :         // la liste des contraintes transversales qui relient les modèles
Résultat :
Cagrég : la liste des contraintes du modèle résultant de l'agrégation de modèle 1 et modèle 2.
1  Début
2    Créer Cres ;
3    Ajouter C1 à Cres ;
4    Ajouter C2 à Cres ;
5    Ajouter Ctrans à Cres ;
6    Pour (i allant de 1 à Connexions.size()) faire
7      Si((Connexions [i].get(1).appliedStereotype('VariableElement')= vrai) et
8         (Connexions [i].get(3).appliedStereotype('VariableElement')= faux)) alors
9         cont ← créerContrainte (Connexions [i].get(1), AtLeastOne);
10        ajouter cont à Cagrég ;
11     Sinon
12        Si ((Connexions [i].get(1).appliedStereotype('VariableElement')= faux) et
13           (Connexions [i].get(3).appliedStereotype('VariableElement')= vrai)) alors
14           cont ← créerContrainte (Connexions [i].get(3), AtLeastOne);
15           ajouter cont à Cagrég ;
16        FinSi ;
17     FinSi ;
18     Si ((Connexions [i].get(1).appliedStereotype('VariableElement')= vrai) et
19        (Connexions [i].get(3).appliedStereotype('VariableElement')= vrai)) alors
20       Si ((occurrence (Connexions [i].get(1), Connexions)=1) et
21          (occurrence (Connexions [i].get(3), Connexions)=1)) alors
22          cont ← créerContrainte ((Connexions [i].get(1), Connexions [i].get(3)), Equivalence);
23          cont1← créerContrainte (Connexions [i].get(1), AtLeastOne);
24          ajouter cont à Cres ;
25          ajouter cont1 à Cres ;
26       Sinon
27         Si (occurrence (Connexions [i].get(1), Connexions) > 1) alors
28           cont ← créerContrainte ((Connexions [i].get(3), Connexions [i].get(1)), Implication);
29           cont1 ← créerContrainte (Connexions [i].get(3), AtLeastOne) ;
30           ajouter cont à Cres ;
31           ajouter cont1 à Cres ;
32         FinSi ;
33         Si (occurrence (Connexions [i].get(3), Connexions) > 1) alors
34           cont ← créerContrainte ((Connexions [i].get(1), Connexions [i].get(3)), Implication);
35           cont1 ← créerContrainte (Connexions [i].get(1), AtLeastOne) ;
36           ajouter cont à Cres ;
37           ajouter cont1 à Cres ;
38         FinSi ;
39       FinSi;
40     FinSi ;
41   Fin ;
42   retourner Cres ;
43 Fin
```

Appliquons l'algorithme AlgorithmeP1 sur l'exemple du modèle assemblé composé des classes composites *Security* et *Camera* tel qu'il est illustré dans la figure 5.10. Dans cet exemple, nous souhaitons agréger les classes composites *Security* et *Camera*. Pour ce faire, l'algorithme prend en entrée toutes les connexions identifiées pendant l'étape d'assemblage. Chaque connexion est définie par le triplet constitué du port lié à la terminaison gauche du connecteur, du connecteur et du port lié à sa terminaison droite. Dans le cas de notre exemple, les quatre connexions identifiées sont les suivantes : (*camControlSec*, *connect1*, *canEnC*), (*camControlSec*, *connect2*, *camInstructC*), (*vFlowSec*, *connect3*, *videoFlowC*) et (*cZoomSec*, *connect4*, *zoomInstructC*). De plus, l'algorithme prend l'ensemble des contraintes incluses dans $C_{Security}$ et C_{Camera} ainsi que l'ensemble des contraintes transversales *Ctrans*. L'algorithme retourne l'ensemble des contraintes globales du modèle suite à l'application des règles d'agrégation.

Le traitement commence par le parcours de la liste des connexions données en entrée et applique les règles d'agrégation selon le cas courant. La première partie du traitement s'intéresse aux connexions qui possèdent un seul port variable. Par exemple la connexion définie par (*vFlowSec*, *connect3*, *videoFlowC*) où le port variable *vFlowSec* est connecté au port obligatoire *videoFlowC* via le connecteur *connect3*. Dans ce cas, une contrainte de type *atleastOne* est associée au port variable *vFlowSec* pour garantir sa présence dans tous les systèmes valides afin d'assurer la connexion. Cette contrainte est ensuite ajoutée à l'ensemble de contraintes globales (lignes 9 et 10). La deuxième partie de l'algorithme traite l'agrégation pour les connexions reliant des ports variables. Dans cette partie, nous distinguons entre les ports variables simples et les ports variables multiples. Par exemple pour la connexion (*cZoomSec*, *connect4*, *zoomInstructC*), les ports *cZoomSec* et *zoomInstructC*sont sont des ports variables simples. Dans ce cas, une première contrainte de type *equivalence* est associée aux deux ports et une deuxième contrainte de type *atLeastOne* est associée au port *cZoomSec* pour indiquer que les ports *cZoomSec* et *zoomInstructC* doivent être présents dans tous les systèmes valides. Dans le cas où un des deux ports est multiple, comme pour les ports de la connexion (*camControlSec*, *connect1*, *canEnC*), une contrainte de type *implication* est associée aux deux ports pour spécifier que la présence du port *canEnC* implique celle du port multiple *camControlSec*. Une deuxième contrainte de type *atLeastOne* est associée au premier port pour qu'il soit présent dans tous les systèmes valides obtenus à partir du modèle d'agrégation résultant.

L'exécution de l'algorithme d'agrégation AlgorithmeP1 sur le modèle assemblé des classes composites *Security* et *Camera* (voir figure 5.10) nous mène au résultat illustré dans la figure 5.11. La figure 5.11 montre l'ajout des contraintes d'agrégation $C_{agrégation}$ qui représente la conjonction des contraintes d'agrégation ajoutées:

$C_{agrégation}$ = *(AtLeastOne(vFlowSec)* ***and*** *AtLeastOne(cZoomSec)* ***and*** *Equivalence(cZoomSec, zoomInstructC)* ***and*** *Implication(camEnC, camControlC)* ***and*** *AtLeastOne(camEnC))*



$C_{Global} = C_{Security}$ ***and*** C_{Camera} ***and*** C_{trans} ***and*** $C_{agrégation}$

Figure 5.11 : Modèle d'agrégation résultant de l'agrégation de Security et Camera(Proposition1)

L'application des règles d'agrégation de la proposition1 implique que tous les systèmes valides obtenus à partir du modèle d'agrégation résultant constituent l'assemblage des systèmes valides obtenus à partir des modèles de lignes de produits en entrée. Autrement dit pour l'exemple de la figure 5.11, les connexions (*camControlSec*, *connect1*, *canEnC*), (*camControlSec*, *connect2*, *camInstructC*), (*vFlowSec*, *connect3*, *videoFlowC*) et (*cZoomSec*, *connect4*, *zoomInstructC*) seront présentes dans tous les systèmes valides obtenus à partir du

modèle d'agrégation résultant. La propriété *(P_{Ag})* est ainsi respectée pour tous les produits valides obtenus.

5.4.2.2. Synthèse

La proposition1 présentée dans cette section impose la présence de toutes les connexions identifiées par le concepteur pendant l'étape d'assemblage. Ceci est réalisé par l'application des règles d'agrégation proposées. L'implémentation de ces règles réalise l'agrégation en respectant la structure composite des modèles. Seulement les éléments externes responsables de la connexion sont traités. L'exécution de cet algorithme se matérialise par la modification de l'information de variabilité de certains éléments de connexions ainsi que par l'ajout d'un ensemble de contraintes d'agrégation associées à ces éléments afin de satisfaire la propriété sémantique de l'agrégation.

5.4.3. Proposition2

La deuxième alternative donne la liberté au concepteur de choisir parmi les connexions identifiées celles qui seront présentes dans les systèmes obtenus à partir du modèle d'agrégation. Ceci implique que seules les connexions choisies par l'utilisateur seront établies et non la totalité des connexions réalisées à l'étape d'assemblage. Pour ce faire, la proposition2 se charge de gérer les ports variables des classes composites représentant les modèles d'entrée ainsi que les connecteurs qui les relient. La proposition2 se base sur un ensemble de règles d'agrégation.

5.4.3.1 Règles d'agrégation :

***Règle 1* :** Si deux ports sont connectables et peuvent être connectés (la connexion est optionnelle), et si l'un des deux ports est variable alors :

- Le connecteur devient variable
- Une contrainte doit être créée pour assurer la présence du port variable si la connexion est établie.

Nous citons l'exemple du port variable *camControlSec* connecté au port obligatoire *camInstructC* via le connecteur *connect2*.

Idem pour l'exemple du port variable *vFlowSec* connecté au port obligatoire *videoFlowC* via le connecteur *connect3*.

Règle 2 **:** Si deux ports sont connectables (hypothèse) et peuvent être connectés, et si les deux ports sont variables alors :

- Le connecteur devient variable
- Une contrainte doit être créée pour assurer la présence des deux ports si la connexion est établie.

Par exemple *cZoomSec* et *zoomInstructC* sont des ports variables connectés par le connecteur *connect4*. De même pour les ports variables *camControSec* et *canEnC* connectés via le connecteur *connect1*.

Algorithme d'agrégation :

Un deuxième algorithme est proposé pour réaliser l'agrégation des modèles de lignes de produits en entrée représentés sous une vue de structure composite. L'algorithme AlgorithmeP2 implémente les règles d'agrégation précédemment décrites pour la proposition2 et montre comment les contraintes d'agrégation sont ajoutées afin d'assurer les connexions entre les ports externes des classes composites de façon à ce que la propriété *(P_{Ag})* soit respectée. Pour illustrer l'exécution de cet algorithme, nous reprenons avec l'exemple de la figure 5.10.

L'algorithme prend en entrée la liste des connexions *Connexions* obtenue à l'issue de la phase d'assemblage. Chaque connexion est définie par les deux ports connectés ainsi que par le lien de connexion choisi pour cette connexion, à l'instar d'un connecteur. *C1* est l'ensemble des contraintes du premier modèle de lignes de produits modèle1. *C2* est l'ensemble des contraintes du deuxième modèle de lignes de produits modèle2. *Ctrans* est l'ensemble des contraintes transversales. *C1*, *C2* et *Ctrans* sont aussi considérés en entrée de l'algorithme d'agrégation. Le résultat est l'ensemble final des contraintes *Cres* obtenu suite à l'application de l'algorithme.

Le traitement commence par l'ajout des listes de contraintes *C1*, *C2* et *Ctrans* à l'ensemble de contraintes global *Cres*. L'algorithme va ensuite parcourir la liste de connexions *Connexions* et appliquer les règles d'agrégation à chacune d'entre elles. Nous identifions deux principales parties du traitement.

La première partie (ligne 7 – ligne19) implémente la première règle d'agrégation où un seul des ports d'une connexion donnée est variable. Donc pour une connexion donnée, si le

premier port est variable (ligne 7) et que le deuxième port est fixe (ligne 8) alors le connecteur qui les relie devient variable (ligne 9). Une contrainte est ensuite ajoutée (lignes 10 et 11) pour assurer que la présence du connecteur implique celle du port variable. Idem, si le premier port est fixe (ligne 13) et que le deuxième port est variable (ligne 14) alors le connecteur qui les relie devient variable (ligne 15). Une contrainte est alors ajoutée (lignes 16 et 17) pour assurer que la présence du connecteur implique la présence du port variable. La deuxième partie (ligne 20 – ligne 28) implante la deuxième règle où les deux ports d'une connexion donnée sont variables. Dans ce cas, le connecteur devient variable et une contrainte est ajoutée pour spécifier que la réalisation de la connexion en question implique la présence des deux connecteurs ensemble. Une fois que toutes les connexions sont traitées, l'algorithme termine son exécution et retourne l'ensemble global des contraintes.

Algorithme P2 : algorithme d'agrégation de la proposition2

Entrée :

Connexions: //la liste de triplets ports connectés et connecteur

C1 : // la liste de contraintes du modèle 1 de la ligne de produits.

C2 : // la liste de contraintes du modèle 2 de la ligne de produits.

Ctrans : // la liste des contraintes transversales qui relient les modèles

Résultat : *Cres :* la liste des contraintes du modèle résultant de l'agrégation de modèle 1 et modèle 2.

1. **Début**
2. Créer *Cres* ;
3. Ajouter *C1* à *Cres* ;
4. Ajouter *C2* à *Cres* ;
5. Ajouter *Ctrans* à *Cres* ;
6. **Pour (**i allant de 1 à *Connexions*.size()**) faire**
7. **Si ((***Connexions*[i].get(1).appliedStereotype ('VariableElement') = vrai) et
8. (*Connexions*[i].get(3).appliedStereotype ('VariableElement') = faux)**) alors**
9. *Connexions*[i].get(2).applyStereotype ('VariableElement') ;
10. cont ← créerContrainte ((*Connexions* [i].get(2), *Connexions* [i].get(1)), Implication);
11. **ajouter** cont à *Cres* ;
12. **Sinon**

13. **Si** ((*Connexions*[i].get(1).appliedStereotype ('VariableElement') = faux) et
14. (*Connexions*[i].get(3).appliedStereotype ('VariableElement') = vrai)) **alors**
15. *Connexions*[i].get(2).applyStereotype ('VariableElement') ;
16. cont ← créerContrainte ((*Connexions* [i].get(2), *Connexions* [i].get(3)), Implication);
17. **ajouter** cont à *Cres*;
18. **FinSi;**
19. **FinSi ;**
20. **Si** ((*Connexions*[i].get(1).appliedStereotype ('VariableElement') = vrai) et
21. (*Connexions*[i].get(3).appliedStereotype ('VariableElement') = vrai)) **alors**
22. *Connexions*[i].get(2).applyStereotype ('VariableElement') ;
23. cont ← créerContrainte ((*Connexions* [i].get(1), *Connexions* [i].get(3)), Equivalence);
24. c ← créerContrainte ((*Connexions* [i].get(2), *Connexions* [i].get(3)), Implication);
25. **ajouter** cont à *Cres* ;
26. **ajouter** c à *Cres* ;
27. **FinSi ;**
28. **Fin ;**
29. **retourner** *Cres* ;
30. **Fin**

Appliquons l'algorithme AlgorithmeP2 sur l'exemple du modèle assemblé composé des classes composites *Security* et *Camera* tel qu'il est illustré dans la figure 5.10. Dans cet exemple, nous souhaitons agréger les classes composites *Security* et *Camera*.

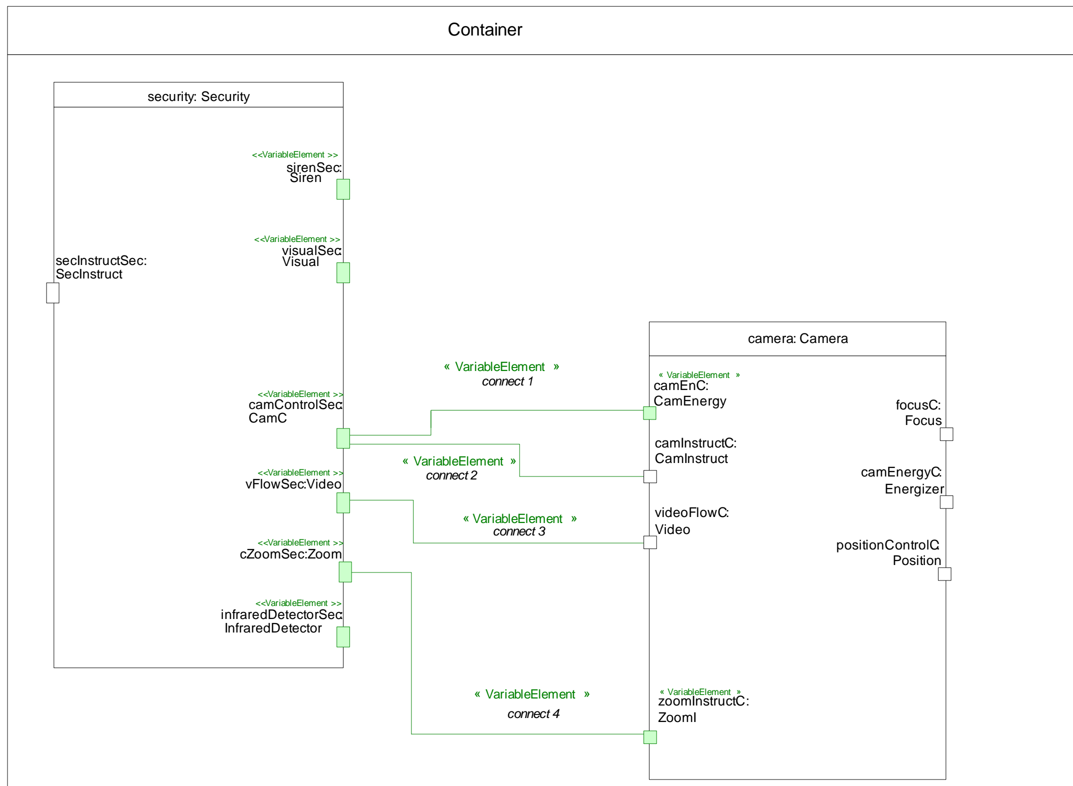
Pour ce faire, l'algorithme prend en entrée toutes les connexions identifiées pendant l'étape d'assemblage. Chaque connexion est définie par le triplet qui est constitué du port lié à la terminaison gauche du connecteur, du connecteur ainsi que du port lié à sa terminaison droite. Dans le cas de notre exemple, les quatre connexions identifiées sont les suivantes : (*camControlSec*, *connect1*, *canEnC*), (*camControlSec*, *connect2*, *camInstructC*), (*vFlowSec*, *connect3*, *videoFlowC*) et (*cZoomSec*, *connect4*, *zoomInstructC*). De plus, l'algorithme prend l'ensemble des contraintes incluses dans $C_{Security}$, C_{Camera} et C_{trans}. L'algorithme retourne l'ensemble des contraintes globales du modèle suite à l'application des règles d'agrégation.

Le traitement commence par le parcour5 de la liste des connexions données en entrée et applique les règles d'agrégation selon le cas courant. La première partie du traitement s'intéresse aux connexions qui possèdent un seul port variable à l'instar de la connexion définie par (*vFlowSec*, *connect3*, *videoFlowC*) où le port variable *vFlowSec* est connecté au port obligatoire *videoFlowC* via le connecteur *connect3*. Dans ce cas, le connecteur *connect3* devient variable (ligne 9) et une contrainte de type *implication* est associée à ce connecteur et au port variable *vFlowSec* pour spécifier que la présence du connecteur *connect3* dans un système valide implique que le port *vFlowSec* soit aussi présent pour assurer cette connexion. Cette contrainte est ensuite ajoutée à l'ensemble de contraintes globales (lignes 10 et 11).

La deuxième partie de l'algorithme traite l'agrégation pour les connexions reliant des ports variables. Par exemple pour la connexion (*cZoomSec*, *connect4*, *zoomInstructC*), les ports *cZoomSec* et *zoomInstructC* sont des ports variables. Dans ce cas, le connecteur *connect4* devient variable (ligne 22) et deux contraintes sont créées pour spécifier que la présence du connecteur *connect4* implique la coprésence des deux ports *cZoomSec* et *zoomInstructC*.

L'exécution de l'algorithme d'agrégation AlgorithmeP2 sur le modèle assemblé des classes composites *Security* et *Camera* (voir figure 5.10) conduit au résultat illustré dans la figure 5.12. La figure 5.12 montre l'ajout des contraintes d'agrégation $C_{agrégation}$ qui représente la conjonction des contraintes d'agrégation suivantes:

$C_{agrégation}$ = *(Implication(connect2, camControlSec)* ***and*** *Implication(connect3, vFlowSec)* ***and*** *Implication(connect4, sZoomSec)* ***and*** *Equivalence(cZoomSec, zoomInstructC)* ***and*** *Implication(connect1, camControlSec)* ***and*** *Equivalence(camControlSec, camEnC))*



$C_{Global} = C_{Security}$ **and** C_{Camera} **and** C_{trans} **and** $C_{agrégation}$

Figure 5.12 : Modèle d'agrégation résultant de l'agrégation de Security et Camera(proposition2)

L'application des règles d'agrégation de la proposition2 permet d'obtenir des systèmes valides à partir du modèle d'agrégation résultant. Ces systèmes constituent l'assemblage des systèmes valides obtenus à partir des modèles de lignes de produits en entrée. Dans l'exemple de la figure 5.12, les connexions (*camControlSec*, *connect1*, *canEnC*), (*camControlSec*, *connect2*, *camInstructC*), (*vFlowSec*, *connect3*, *videoFlowC*) et (*cZoomSec*, *connect4*, *zoomInstructC*) peuvent être choisies par l'utilisateur pour être présentes dans tous les systèmes valides obtenus à partir du modèle d'agrégation résultant. Cependant, elles peuvent aussi être absentes dans un ou plusieurs produits valides. La satisfaction de la propriété (P_{Ag}) est ainsi dépendante des choix de l'utilisateur pour les connexions qu'il veut avoir dans les produits valides.

La complexité (au pire cas) des deux algorithmes d'agrégation (proposition1 proposition2) présentés dans cette sous section est linéaire (O (n)) où n représente le nombre de connexions identifiées pendant l'étape d'assemblage. Le nombre maximal de n est atteint quand toutes les connexions sont parcourues et traitées. Le tableau 5.1 présente les caractéristiques des deux algorithmes en termes de complexité et de terminaison (algorithme ne bouclant pas infiniment).

	AlgorithmeP1	**AlgorithmeP2**
Calcul de complexité	2 boucles imbriquées : Une boucle de n itérations et qui contient un appel d'une fonction *(occurence)* qui inclut une autre boucle imbriquée de n itérations. → O(n^2) **Complexité totale → O(n^2)**	Une boucle de n itérations au total. **Complexité totale → O(n)**
Absence de boucles infinies	Présence de 2 boucles dont le nombre d'itérations est connu à l'avance.	Présence de 1 boucle dont le nombre d'itérations est connu à l'avance.

Tableau 5.1 – Les caractéristiques des 2 algorithmes d'agrégation en termes de complexité et de terminaison

Il ne s'agit pas d'une preuve par récurrences pour la terminaison ni d'un calcul détaillé de la complexité. Toutefois, il est important de prendre en considération ces caractéristiques et de s'assurer que ces algorithmes répondent bien au but de l'utilisateur qui est l'agrégation de modèles de lignes de produits.

5.4.3.2. Synthèse

À l'encontre de la proposition1, la proposition2 présentée dans cette section ne force pas la présence de toutes les connexions identifiées par le concepteur pendant l'étape d'assemblage. Pour chacun des systèmes valides obtenus à partir du modèle d'agrégation, la proposition2 lui donne la liberté de sélectionner les connexions qui seront présentes dans le système. Dans le cas où le concepteur ne sélectionne aucune connexion, la propriété sémantique de l'agrégation

ne sera pas respectée. Nous estimons dans ce cas que la satisfaction de cette propriété est de la responsabilité du concepteur.

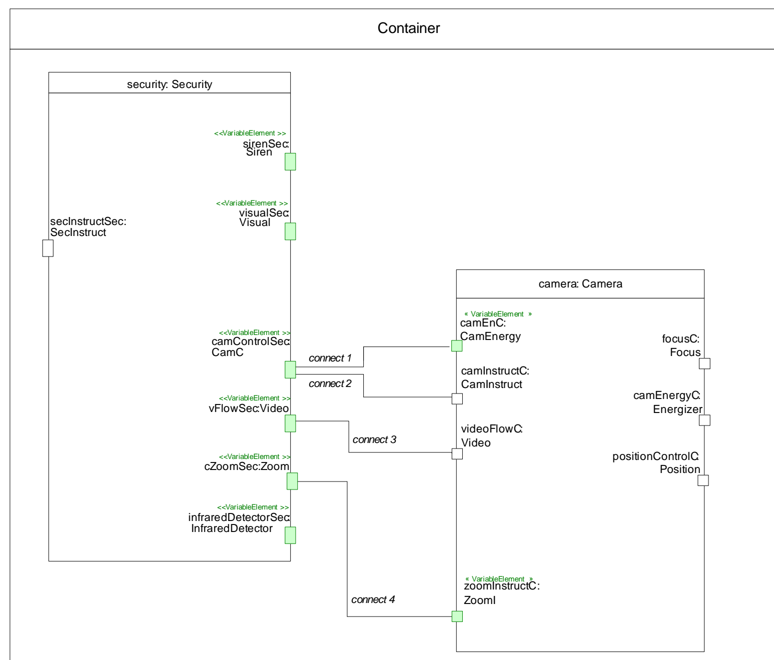
La proposition2 est réalisée par l'application des règles d'agrégation présentées dans la sous-section 5.4.3.1. L'implémentation de ces règles réalise l'agrégation en respectant la structure composite des modèles. Seuls les éléments externes responsables de la connexion sont traités. L'exécution de cet algorithme se matérialise par la modification de l'information de variabilité de certains éléments de connexions ainsi que par l'ajout d'un ensemble de contraintes d'agrégation associées à ces éléments afin de satisfaire la propriété sémantique de l'agrégation.

5.4.4. Agrégation et effets de bord

Les propositions d'agrégation de modèles de lignes de produits présentées dans les sous-sections 5.4.2.1 et 5.4.3.1 ont un impact direct sur les systèmes valides obtenus à partir du modèle d'agrégation résultant. Ceci est dû principalement aux contraintes transversales ainsi qu'aux contraintes ajoutées durant l'étape d'agrégation. Prenons l'exemple de la proposition1 qui impose que toutes les connexions identifiées par l'utilisateur dans l'étape d'assemblage soient présentes dans tous les systèmes obtenus à partir du modèle d'agrégation résultant. Ceci implique que certains éléments variables ne pourront plus être présents dans les systèmes obtenus du modèle d'agrégation. Par conséquent, le nombre de systèmes valides obtenus à partir du modèle d'agrégation va diminuer par rapport au nombre de systèmes valides obtenus avant l'étape d'agrégation. Plus les éléments variables qui assurent les connexions ont des contraintes de variabilité qui les relient au reste des éléments variables internes des modèles d'entrée, plus l'impact de l'agrégation sera important sur le nombre de systèmes obtenus du modèle d'agrégation. Les mêmes effets peuvent être constatés lors de l'utilisation de la proposition2.

Un autre effet de bord est susceptible d'apparaitre si les contraintes d'agrégation ajoutées sont en contradiction avec les contraintes de variabilité associées aux modèles avant l'agrégation. Dans ce cas, aucun système ne pourra être obtenu à partir du modèle d'agrégation résultant. Supposons que nous avons le modèle d'assemblage de la figure 5.13 où $C_{Security}$ inclut la contrainte suivante par exemple : $Cont_1$= *xor(camControlSec, vFlowSec, cZoomSec).* Cette contrainte spécifie que si l'un des ports *camControlSec, vFlowSec* et *cZoomSec* est présent dans un des systèmes valides alors aucun des autres ports ne doit être présent dans le même système valide. D'autre part, sachant que la proposition1 impose la présence des ports

camControlSec, vFlowSec et *cZoomSec*, les contraintes d'agrégation seraient contradictoires avec la contrainte $Cont_1$. Par conséquent, l'ensemble de systèmes valides obtenu à partir de ce modèle d'agrégation est l'ensemble vide.



$C_{Global} = C_{Security}$ ***and*** C_{Camera} ***and*** C_{trans} ***and*** $C_{agrégation}$

Figure 5.13 : Modèle d'agrégation et contraintes contradictoires

5.5. Conclusion

Le mécanisme d'agrégation permet de combiner des modèles de lignes de produits ne présentant pas de similarités mais qui peuvent être liés par des contraintes transversales. L'agrégation permet de gérer la variabilité des éléments d'interface connectés. La description du processus d'agrégation contient deux phases dont la première s'intéresse à la consolidation des modèles en entrée alors que la seconde réalise leur agrégation. A l'issue de la première phase, nous obtenons des modèles de lignes de produits enrichis en termes de contraintes structurelles. Ces contraintes permettent d'assurer l'obtention de modèles de produits structurellement cohérents et complets. La seconde phase du processus représente la phase la plus importante puisqu'elle réalise l'agrégation des modèles. La première étape de cette phase, appelée étape d'assemblage, concerne le concepteur des modèles qui a pour tâche d'identifier et d'établir les connexions entre les classes composites des modèles à agréger. Une fois les connexions identifiées, nous procédons à l'étape d'agrégation des modèles assemblés. Nous présentons deux propositions dont la première a pour but de gérer la variabilité des éléments de connexions tels que toutes les connexions identifiées doivent être

présentes dans tous les modèles de produits obtenus à partir du modèle d'agrégation résultant. Un ensemble de règles d'agrégation est donc proposé pour satisfaire cette proposition. L'application de ces règles a pour conséquence l'ajout des contraintes d'agrégation afin d'imposer la présence de toutes les connexions dans tous les modèles de produits valides. La deuxième proposition donne plus de liberté au concepteur. La présence de la connexion dans un modèle de produits valide est déterminée par le concepteur. Un ensemble de règles d'agrégation est donc proposé pour satisfaire cette proposition. Des contraintes d'agrégation sont ajoutées pour identifier les éléments structurels nécessaires pour la présence d'une ou de plusieurs connexions choisies par le concepteur pour être présentes dans un modèle de produit valide. Nous avons aussi présenté certains effets de bord de l'agrégation. Ces effets de bord peuvent être évités dans certains cas en affectant par exemple des priorités aux contraintes existantes. Cependant, la réponse à une telle question reste ouverte pour de prochains travaux. Dans le chapitre suivant, nous présenterons une évaluation des contributions proposées dans ce manuscrit.

CHAPITRE 6
Evaluation

6.1. SEQUIOA : Environnement pour le développement des lignes de produits logiciels.... 114
6.1.1. Processus de développement SEQUOIA.... 114
6.1.2. Architecture 115
6.1.2.1. Outil de propagation 116
6.1.2.3 Outil de calcul de produits 117
6.1.2.4. Outil de génération de modèle de décision 117
6.1.2.5. Outil de dérivation 118
6.2. Implémentation.... 119
6.2.1. Outil de consolidation.... 119
6.2.2. Outil de fusion 122
6.3. Consolidation de modèles de lignes de produits logiciels : Evaluation et apports.... 124
6.3.1. Stratégie d'évaluation.... 124
6.3.2. Résultats.... 125
6.4. Fusion de modèles : Evaluation et apports.... 128
6.4.1. Stratégie d'évaluation.... 128
6.4.2. Résultats.... 128
6.5. Conclusion.... 129

Ce chapitre a pour but d'évaluer certaines contributions précédemment présentées dans ce manuscrit. Il est structuré en quatre sections: la première présente l'environnement de développement des lignes de produits logiciels, Sequoia, dans lequel les contributions sont intégrées. La deuxième section donne plus de détails sur l'implémentation de la consolidation et de la fusion. La troisième section se focalise sur les résultats obtenus grâce à l'outil de consolidation implémenté. Enfin, la quatrième section a pour objectif d'analyser les résultats obtenus pour l'outil de fusion.

6.1. SEQUIOA : Environnement pour le développement des lignes de produits logiciels

Dans cette section, nous présentons l'environnement Sequoia. Cet environnement est dédié au développement des lignes de produits logiciels. Sequoia propose un processus complet pour le développement des lignes de produits logiciels allant de la capture des exigences jusqu'à la génération des produits finaux concrets. Sequoia propose aussi une chaîne d'outils où chaque phase du processus de développement est réalisée grâce à un outil de la chaîne. Nos contributions seront intégrées dans cet environnement. Pour cela, nous présentons le processus Sequoia ainsi que l'ensemble d'outils qu'il inclut.

6.1.1. Processus de développement SEQUOIA

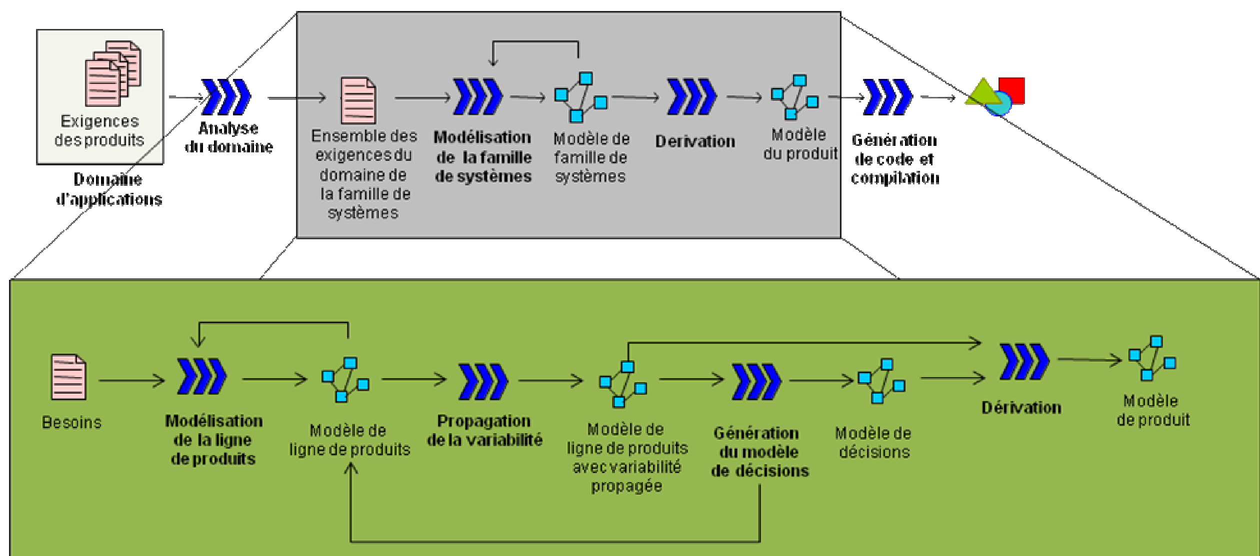


Figure 6.1 : Processus de développement de lignes de produits logiciels

Le processus proposé par Sequoia pour le développement des lignes de produits logiciels se base sur l'ingénierie dirigée par les modèles (IDM), donc sur un ensemble de transformations de modèles. Ces transformations sont réalisées sur plusieurs phases durant lesquelles les modèles sont raffinés jusqu'à la réalisation concrète du logiciel.

En effet, le processus prend comme entrée l'ensemble des exigences des produits envisagés constituant le domaine. Ces exigences sont ensuite analysées et regroupées de façon à identifier celles qui représentent les points communs entre tous les produits de la ligne de produits logiciels et les exigences représentant les options dont la présence dépend du produit envisagé.
La structure de la ligne de produits est obtenue à l'issue de la phase de modélisation. Elle consiste à construire le modèle de la ligne de produits à partir des exigences de domaine. Cette structure doit donc inclure la modélisation des points communs ainsi que les options précédemment identifiés (voir profil Sequoia dans annexe A). La phase suivante, nommée phase de dérivation, consiste à effectuer des choix parmi les options présentes dans le modèle de la ligne de produits. Ces options feront partie du modèle de produit envisagé. La dernière phase consiste à générer et compiler du code à partir du modèle obtenu afin d'aboutir au produit concret.

6.1.2. Architecture

L'environnement Sequoia est composé d'un ensemble d'outils implémentés sous formes de plugins Eclipse (voir la figure 6.2). Pour modéliser les lignes de produits logiciels, nous utilisons le modeleur Papyrus [3].

[3] http: //www.eclipse.org/modeling/mdt/papyrus/

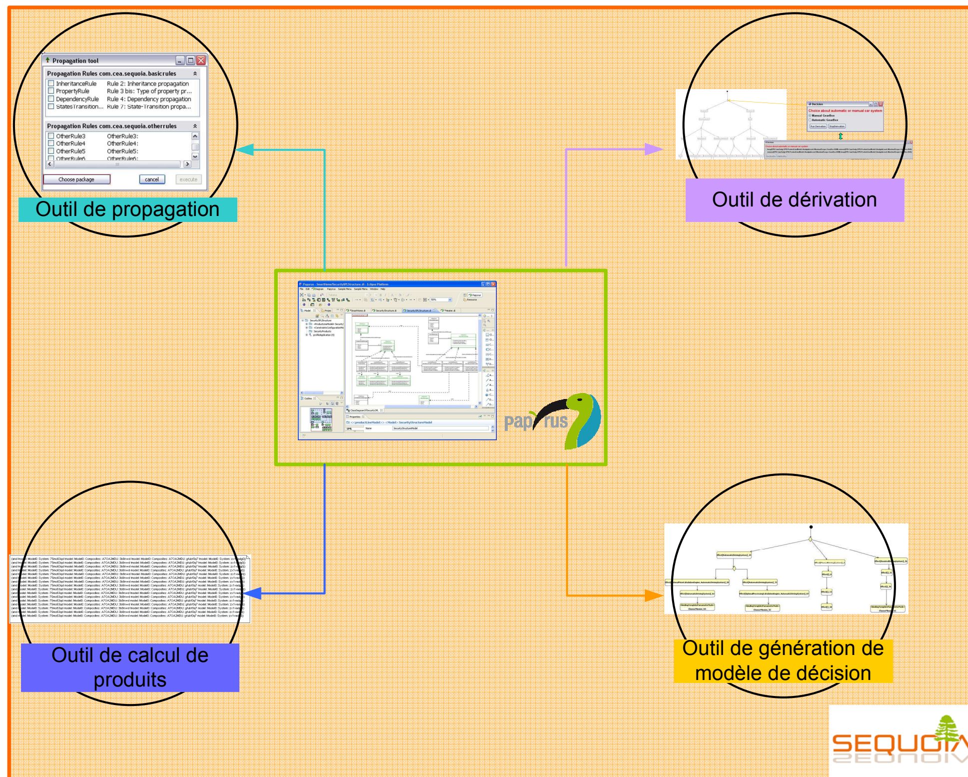


Figure 6. 2: Outils de Sequoia

6.1.2.1. Outil de propagation

Après avoir spécifié les éléments variables du modèle de ligne de produits logiciels, il est important de s'assurer de la cohérence des éléments. Cette cohérence est indispensable pour la génération de modèles de produits bien formés, c'est-à-dire conforme à la sémantique UML. Pour obtenir tel résultat, il faut déterminer l'impact de la variabilité à travers le modèle. En effet, définir un élément comme variable peut affecter par construction d'autres éléments du modèle et rendre ces derniers implicitement variables.

C'est ainsi que vient le rôle de l'outil de propagation de variabilité. Le traitement réalisé par cet outil consiste donc à rendre implicitement variables tous les éléments qui sont affectés par des éléments explicitement variables. Les éléments variables propagés seront identifiés à partir des éléments variables spécifiés par l'utilisateur. Ajoutés à cette identification, des contraintes de variabilité dédiées définissent ces liens et assurent la cohérence en termes de variabilité dans le modèle complet de la ligne de produits logiciels. En général, l'ajout automatique de la variabilité sur un élément provient du fait qu'un autre élément dans le modèle est variable. La contrainte qui relie les éléments est en général de type implication.

Le traitement assuré par l'outil de propagation se base sur des règles dites « basiques » qui ont été extraites de l'analyse du méta-modèle UML 2.0. Ces règles sont non exhaustives et proviennent de l'étude de la structuration des éléments dans le diagramme de classes et du

diagramme de machines à états de protocole. Cependant, la propagation de la variabilité dans un modèle de ligne de produits logiciels peut provoquer une explosion du nombre d'éléments variables dans un modèle. Cette explosion rend difficile l'exploitation et l'analyse de la dérivation. Afin de limiter cela, certaines règles sont modifiées.

La conception d'une ligne de produits logiciels ne se limite pas seulement qu'à l'utilisation du langage UML mais aussi à l'utilisation conjointe de profils UML complémentaires tels que par exemple SysML[4] et MARTE[5]. Par conséquent, l'outil de propagation permet d'ajouter des règles liées au métier des concepteurs, et de supprimer l'application de règles basiques.

6.1.2.3 Outil de calcul de produits

Afin de générer le guide d'aide à la dérivation. Il faut tout d'abord être capable de calculer tous les produits valides sachant qu'il est possible de transformer ce problème en un problème de satisfaction de contraintes booléennes à partir de l'ensemble de contraintes provenant des groupes de variation.

Les groupes de variation expriment des contraintes sur l'existence des éléments. Les produits possibles sont l'ensemble des combinaisons booléennes respectant les contraintes. En utilisant un solveur de type SAT, un solveur des satisfactions de contraintes booléenne [62], il est possible d'obtenir l'ensemble des combinaisons possibles. Grâce à l'ensemble des combinaisons possibles, il est possible de construire le modèle de décisions.

6.1.2.4. Outil de génération de modèle de décision

L'outil de génération du modèle de décisions de Sequoia génère automatiquement le modèle de décisions à partir de l'analyse systématique du modèle de ligne de produits logiciels. La cohérence est donc implicite par construction. Le modèle de décisions est présenté sous forme d'un diagramme d'activité UML. Il propose une séquence de choix pour la dérivation du modèle de ligne de produits logiciels. Chaque choix effectué impliquera un ensemble

[4]*SysML*, Object Management Group, 2010.

[3]*The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems*, Object Management Group, 2003.

d'actions ou effets à réaliser sur le modèle de la ligne de produits logiciels pour obtenir le modèle d'un produit particulier.

6.1.2.5. Outil de dérivation

Chaque chemin possible du modèle de décisions correspond à la réalisation d'un modèle de produit. La dérivation d'un modèle de produit est facile à réaliser. L'outil de dérivation analyse le modèle de décisions et permet de collecter les séquences d'effets de chaque chemin. Sachant que chaque effet contient les instructions de transformation de modèles, la séquence d'effets contenue dans un chemin correspond à la séquence de transformations de modèle nécessaire pour la dérivation. La dérivation consistera alors à faire :

- des suppressions d'éléments,
- des substitutions de paramètres.

Arrivée à l'état final, il faut réaliser un nettoyage du modèle qui passera par :

- la suppression des éléments du modèle liés au calcul de la dérivation (suppressions des états isolés…),
- la dés-application du profil Sequoia,
- la suppression du modèle de décisions

L'outil de dérivation existant de Sequoia est au stage de prototype. Il propose à l'utilisateur les choix à effectuer de manière interactive. Pour cela, il interprète le modèle de décisions, à partir du nœud initial jusqu'au nœud final. Lorsque qu'il arrive sur un nœud de décision, l'outil de dérivation propose de choisir parmi les résolutions spécifiées par le nœud résolution.

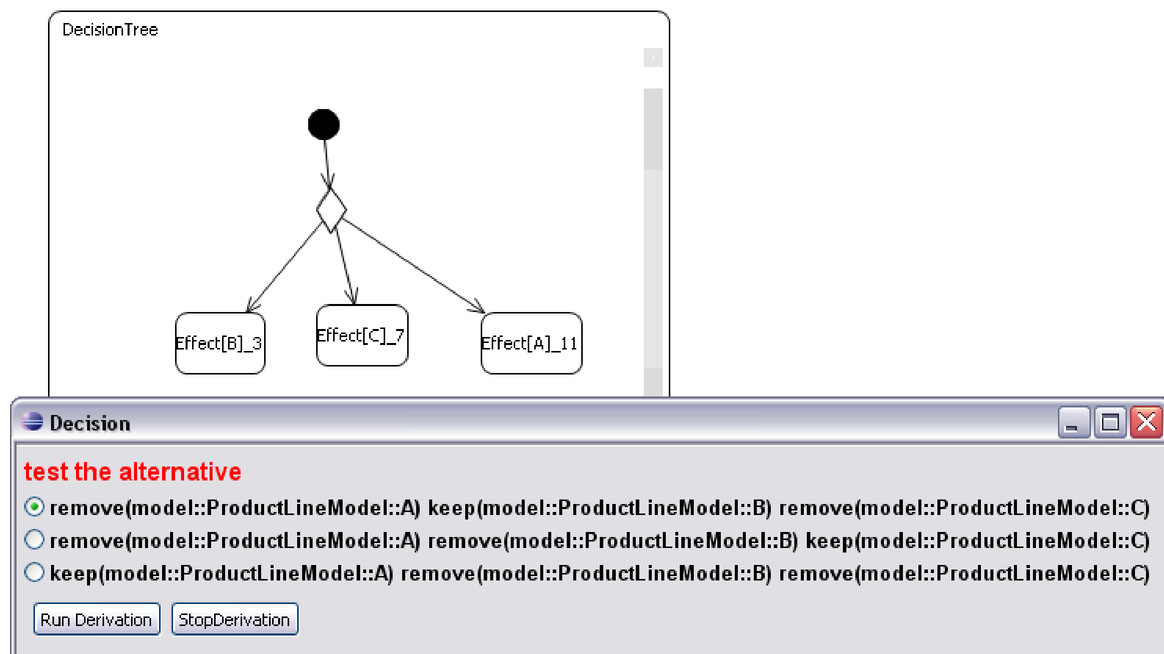


Figure 6.3 : Dérivation sur un nœud Décision

La figure 6.3 montre l'exécution de l'outil de dérivation qui interprète le modèle de décisions issu d'un modèle. Comme illustré, l'outil teste une contrainte alternative sur trois éléments. En rouge est indiqué la motivation de la contrainte (ici le test de la contrainte alternative) et en dessous les 3 possibilités que le concepteur peut choisir. A tout moment le concepteur peut arrêter la dérivation. Dans ce cas nous obtenons un modèle de ligne de produits logiciels partiellement dérivé.

Nous avons ainsi présenté dans cette section le processus de développement de Sequoia ainsi que les différents outils implémentés dans ce cadre. Nous rappelons que les outils qui implémentent les propositions de cette thèse sont intégrés dans l'environnement Sequoia. Nous expliquons dans la section suivante comment l'outil de consolidation et l'outil de fusion sont implémentés.

6.2. Implémentation

Dans cette section, nous exposons les informations d'implémentation des outils de consolidation et de fusion.

6.2.1. Outil de consolidation

L'outil de consolidation des modèles de lignes de produits logiciels implémente les règles de consolidation précédemment présentées dans le chapitre 3. Nous rappelons que ces règles s'appliquent sur les modèles de lignes de produits logiciels représentés sous formes de

structures composites afin d'assurer l'obtention de modèles de produits structurellement cohérents et complets. Ces règles se matérialisent par l'ajout de contraintes structurelles qui permettent d'éviter certaines "anomalies". Nous rappelons les règles de consolidation:

(Reg1) : Règle de connexion à un port simple

Si deux ports simples sont connectés et qu'au moins un des deux est variable, alors le deuxième port doit aussi être variable. Une contrainte de type équivalence doit être créée entre les deux ports pour spécifier que si l'un des deux ports est présent dans un modèle de produit dérivé alors l'autre le sera aussi et inversement.

(Reg2) : Règle de connexion à un port multiple

Si un port est connecté à un autre port multiple et qu'au moins un des deux est variable, alors le deuxième port doit aussi être variable. Une contrainte de type implication doit être créée pour spécifier que la présence du premier port dans un modèle de produit dérivé implique la présence du port multiple.

(Reg3) : Règle de port variable attaché sur une part variable

Si un port est variable et qu'il est attaché sur une part variable, une contrainte de type implication doit être créée pour spécifier que la présence du port implique la présence de la part sur laquelle il est attaché.

(Reg4) : Règle de port variable connecté à un port attaché sur une part variable

Si un port variable est connecté à un port qui est attaché sur une part variable, alors une contrainte de type implication est créée pour spécifier que la présence du premier port implique la présence de la part.

(Reg5) : Règle de connecteur variable reliant des ports simples

Si un connecteur variable relie deux ports dont au moins un est variable alors, le port obligatoire devient variable. Une contrainte de type équivalence doit être créée pour spécifier la coprésence du connecteur et des deux ports qu'il relie si un des trois est présent dans un modèle.

(Reg6) : Règle de connecteur variable reliant deux ports dont au moins un est multiple

Si un port est connecté à un autre port multiple via un connecteur variable et qu'au moins un des deux ports est variable, alors le deuxième le devient aussi. Une contrainte de type implication doit être créée pour spécifier que la présence du connecteur dans un modèle de produit implique la présence du premier port et que la présence de ce dernier implique la présence du port multiple.

L'outil de consolidation implémente ces règles dans un plugin Eclipse. Cet outil est implémenté en utilisant Papyrus, le modeleur basé sur Eclipse afin de définir les structures composites UML2 des lignes de produits logiciels. Les figures 6.4 et 6.5 montrent l'utilisation de l'outil de consolidation sur un modèle de ligne de produits logiciels.

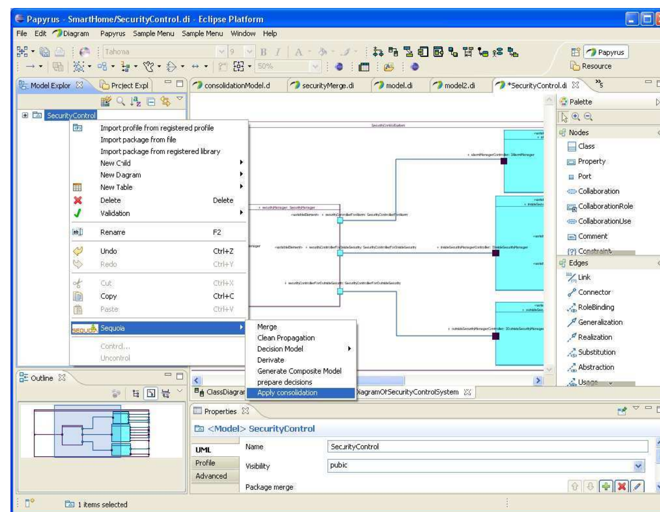


Figure 6. 4: Menu de l'outil consolidation

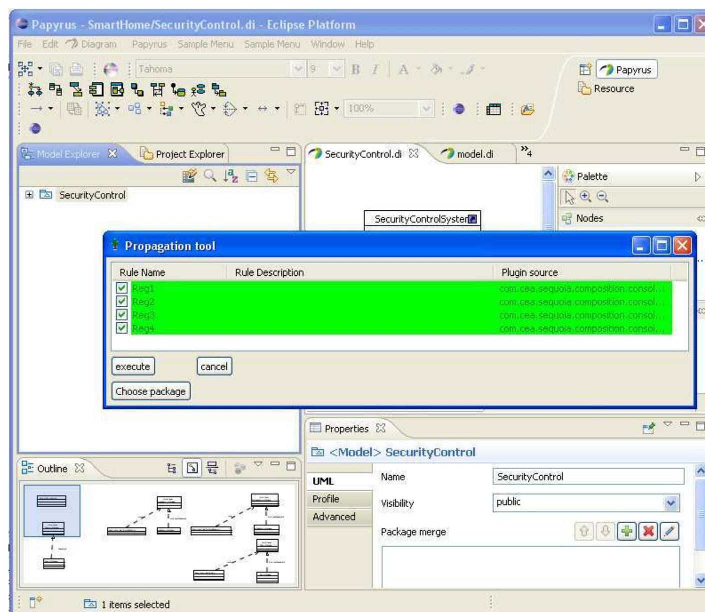


Figure 6. 5: Choix des règles de consolidation

L'outil prend en entrée le modèle de ligne de produits logiciels modélisé sous une vue de structure composite. Les règles de consolidation sont ensuite appliquées et génèrent un ensemble des contraintes structurelles. Les contraintes générées sont ajoutées aux contraintes initiales et peuvent ainsi être visualisées sur l'éditeur de modèles de Papyrus.

L'utilisation de l'outil de consolidation n'est pas limitée au contexte de composition. Il peut être utilisé de manière indépendante pour toute structure composite de ligne de produits logiciels en vue d'utilisation. Par exemple, l'outil de consolidation peut bien être utilisé après la phase de modélisation de la ligne de produits logiciels comme indiquée dans le processus Sequoia illustré dans la figure 6.1.

6.2.2. Outil de fusion

L'outil de fusion réalise la fusion des éléments structurels en exécutant les règles de fusion présentées dans la section 4.4 du chapitre 4 (voir algorithmes 1.a et 1.b). Il implémente aussi l'algorithme de fusion des contraintes tel qu'il est décrit dans la même section du chapitre 4 (voir algorithme 2). L'outil de fusion est implémenté dans un plugin Eclipse en utilisant Papyrus, le modeleur basé sur Eclipse afin de définir les structures composites UML2 des lignes de produits logiciels à fusionner. La figure 6.6 montre l'utilisation de l'outil de fusion sur les modèles de ligne de produits logiciels.

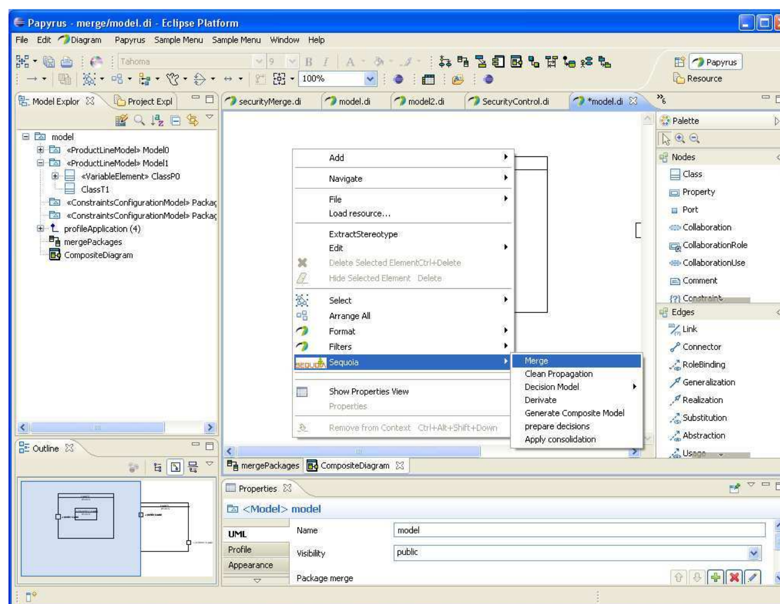


Figure 6.6 :Menu de l'outil de fusion

L'outil prend en entrée les modèles de lignes de produits logiciels à fusionner. Chacun de ces modèles consiste en un modèle structurel qui contient les éléments structurels à fusionner ainsi que l'ensemble des contraintes de variabilité. L'exécution de l'outil de fusion génère le

modèle de ligne de produits résultant de la fusion. Ce modèle contient la structure composite résultante ainsi que le résultat des contraintes fusionnées. Dans la figure 6.7, le paquetage RESULT contient le modèle structurel résultant de la fusion et le paquetage RES_CONSTRAINTS inclut les contraintes fusionnées.

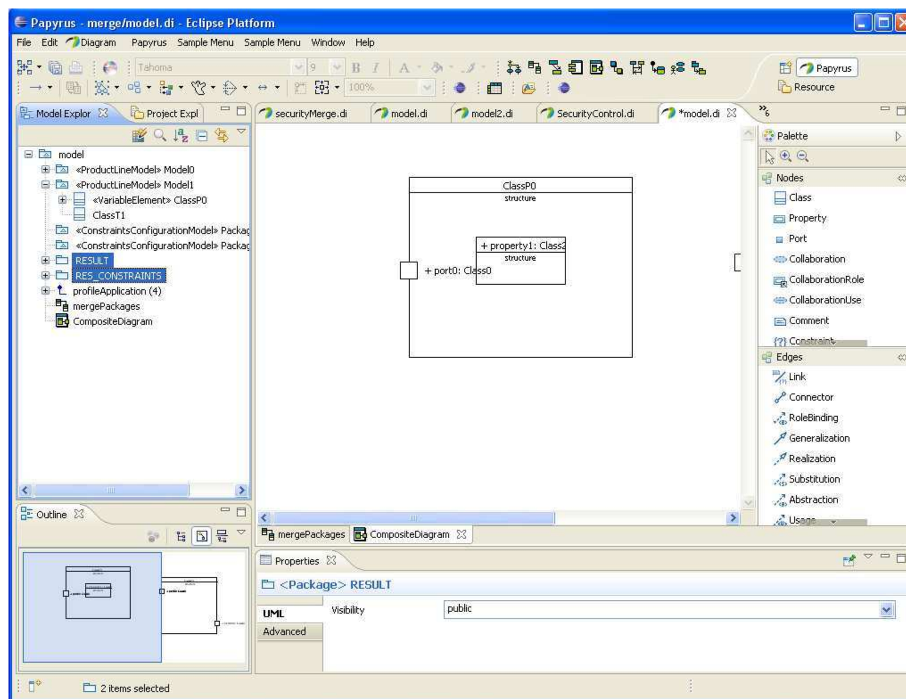


Figure 6. 7: Résultat de fusion sur l'éditeur de modèles

L'outil de fusion permet ainsi de fusionner différents fragments de modèles de lignes de produits logiciels développés séparément. Il peut être utilisé pour d'autres raisons comme par exemple la fusion de deux versions de modèles de lignes de produits logiciels évolués.

6.3. Consolidation de modèles de lignes de produits logiciels : Evaluation et apports

Dans cette section, nous présentons quelques résultats d'évaluation de l'outil de consolidation implémenté. Pour ce faire, nous avons défini comme critère d'évaluation le nombre de modèles de produits incomplets éliminés grâce à l'exécution des règles de consolidation implémentées dans cet outil.

6.3.1. Stratégie d'évaluation

Nous avons montré dans le chapitre 3 comment les règles de consolidation permettent d'éviter l'obtention de produits structurellement incomplets. Pour appuyer cette proposition, nous proposons des étapes d'évaluation bien définies. La stratégie d'évaluation de l'outil de consolidation de modèles de lignes de produits logiciels fixe comme critère d'évaluation le nombre de produits incomplets éliminés. Pour ce faire, nous avons procédé comme décrit dans la figure 6.8. Cette figure illustre le déroulement chronologique des activités d'évaluation de l'outil de consolidation implémenté.

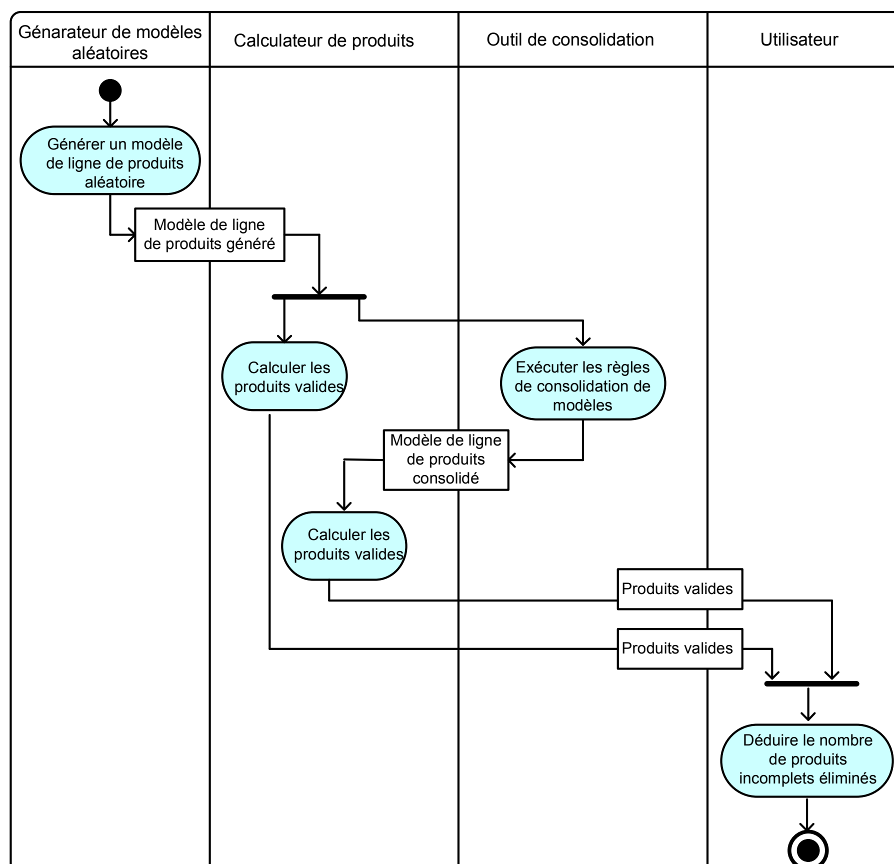


Figure 6.8 : Diagramme d'activité de la démarche d'évaluation de l'outil de consolidation

La première activité consiste à générer aléatoirement un modèle de ligne de produits logiciels. Un générateur de modèle aléatoire a été implémenté pour générer des modèles de lignes de

produits logiciels sous une vue de structure composite. Ce générateur est paramétrable en ce qui concerne le nombre total d'éléments du modèle, le pourcentage d'éléments variables, le nombre maximal de *parts* par classe et le nombre maximal de *ports* par *parts*. Ainsi le modèle de ligne de produits logiciels est généré sous une vue de structure composite et porte des annotations de variabilité (stéréotype *«VariableElement»*) sur certains éléments.

La deuxième activité s'intéresse à calculer les produits valides à partir du modèle de ligne de produits généré lors de la première activité. Le calculateur de produits est responsable de la réalisation de cette activité. A l'issue de cette activité nous obtenons l'ensemble des produits valides du modèle généré lors de la première activité.

Parallèlement à la deuxième activité, une consolidation est exécutée sur le modèle de ligne de produits logiciels généré lors de la première activité. Cette activité est assurée par l'outil de consolidation qui exécute l'ensemble des règles de consolidation précédemment décrites. Nous rappelons que l'exécution des règles de consolidation peut se matérialiser par l'augmentation du nombre d'éléments variables ainsi que par l'ajout de contraintes structurelles sur les éléments variables. A l'issue de cette activité, nous obtenons un modèle de lignes de produits logiciels consolidé. Par la suite le calculateur de produits va calculer l'ensemble des produits valides à partir du modèle de ligne de produits logiciels consolidé. L'ensemble des produits obtenus à l'issu de cette étape ne contient pas des produits structurellement incomplets. Il est alors comparé avec l'ensemble de produits calculés à partir du modèle de ligne de produits logiciels non consolidé. C'est ainsi que la différence entre les deux ensembles de produits obtenus nous permet de déduire le nombre de produits incomplets éliminés suite à l'exécution des règles implémentées dans l'outil de consolidation.

6.3.2. Résultats

Pour mener ce travail d'évaluation, nous avons varié le nombre de ports et de parts existants dans le modèle de lignes de produits logiciels à consolider de 5 à 25. Pour chaque valeur donnée, 50% des ports et parts sont annotés comme variables. Nous avons ensuite évalué comment le nombre de ports et parts variables peut affecter le nombre de produits incomplets éliminés.

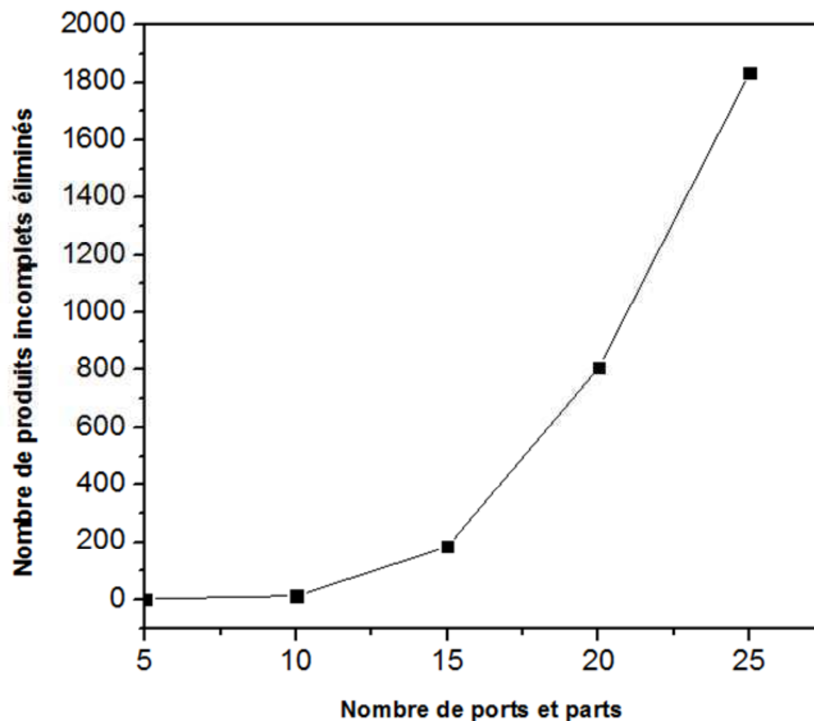


Figure 6. 9: Courbe d'évaluation de l'outil de consolidation

La courbe illustrée dans la figure 6.9 montre les résultats expérimentaux obtenus. En effet, le nombre de produits incomplets éliminés augmente de manière proportionnelle au nombre de ports et parts variables dans le modèle de ligne de produits logiciels. Par exemple pour 20 ports et parts dont 50% sont variables, le nombre de produits incomplet éliminés grâce à l'application des règles de consolidation s'élève à plus de 800 produits. Pour certains produits éliminés, nous avons constaté durant l'évaluation qu'un même produit peut être éliminé par plus d'une seule règle de consolidation. Ceci s'explique par le fait que ce produit présente deux anomalies détectées par deux règles différentes. L'application des règles de consolidation implique alors que ce produit soit éliminé grâce aux deux règles qui ont détecté les anomalies. Prenons l'exemple de la figure 6.10 qui présente la classe composite *Security* représentant une ligne de produits qui ne possède aucune contrainte structurelle associée à ses éléments (figure 6.10.(a)) ainsi qu'un modèle de produit dérivé (6.10.(b)).

Dans la figure 6.10.(a) le port *siren* est variable et attaché sur la part variable *alarm*. Il est connecté au port variable *sirenSec* via un connecteur. Nous pouvons remarquer l'absence de toute contrainte structurelle qui relie le port *siren* à la part d'*alarm* ou au port *sirenSec*, (voir la figure 6.10.(a)). La dérivation du modèle de ligne de produits, illustré dans 6.10.(a), montre l'obtention du modèle de produit présenté dans la figure 6.10.(b). La figure présente un modèle de produits dans lequel le port *siren* doit être présent et ce inversement aux cas de

alarm et *sirenSec*. Cependant, la figure 6.10.(b) montre l'absence de *siren*, *sirenSec* et *alarm*. Ceci s'explique par le fait que le port *siren* ne peut être visualisé et sa présence ne peut avoir un sens que si la part *alarm* est elle aussi présente dans le modèle. L'absence de contrainte structurelle qui relie le port *siren* à la part *alarm* sur laquelle il est attaché peut engendrer des modèles incomplets comme dans la figure 6.10.(b). De même pour l'absence du port *sirenSec*. Ceci s'explique par l'absence de contrainte structurelle qui relie les deux ports *siren* et *sirenSec*. Nous remarquons ainsi que le modèle de produit obtenu dans la figure 6.10.(b) présente deux anomalies dont la première est détectée par *(Reg1)* et la deuxième est détectée par *(Reg3)*.

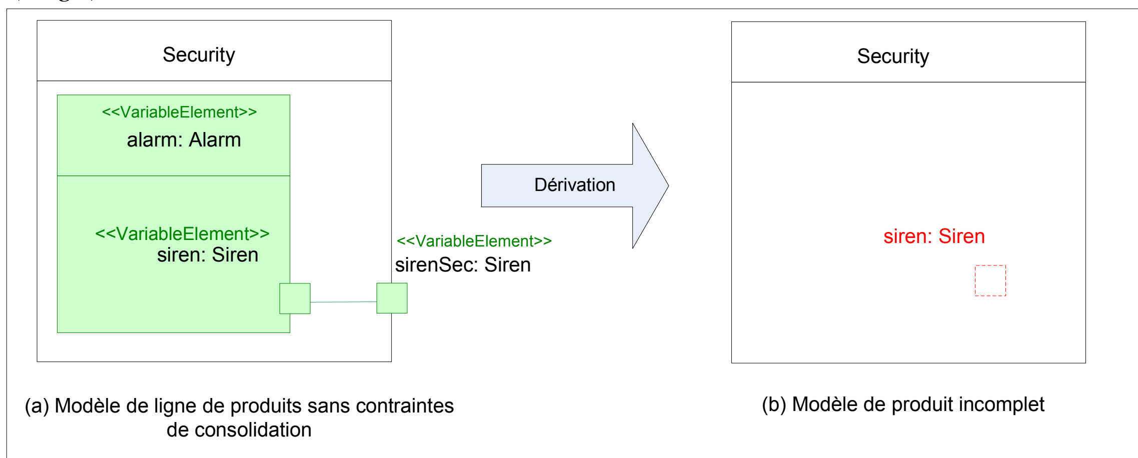


Figure6.10 : Avant application des règles de consolidation *(Reg1)* et *(Reg3)*

La figure 6.11 montre l'application de *(Reg1)* par ajout de la contrainte d'équivalence associée aux ports *siren* et *sirenSec* (voir figure 6.11.(a)). La contrainte $Cont_1$ impose la coprésence des deux ports, dans le cas où un des deux doit être présent dans le modèle de produits dérivé. Cette contrainte assure que le modèle incomplet de la figure 6.10.(b) ne peut plus être obtenu. De même la figure 6.11 montre l'application de *(Reg3)* par ajout de la contrainte d'implication associée aux port *siren* et à la part *alarm* (voir figure 6.11.(a)). La contrainte $Cont_2$ impose la présence de la part *alarm* si le port *siren* est présent. Comme la contrainte $Cont_2$, cette contrainte empêche l'obtention du modèle incomplet de la figure 6.10.(b). Ainsi chacune des règles de consolidation *(Reg1)* et *(Reg3)* permettent d'éliminer le modèle de produit incomplet de la figure 6.11.(b). Le modèle de produits obtenu dans la figure 6.11.(b) après application de *(Reg1)* et *(Reg3)* est complet et contient les deux ports *siren* et *sirenSec* et la part *alarm* grâce à l'ajout des contraintes structurelles $Cont_1$ et $Cont_2$ comme le montre la figure 6.11.(a).

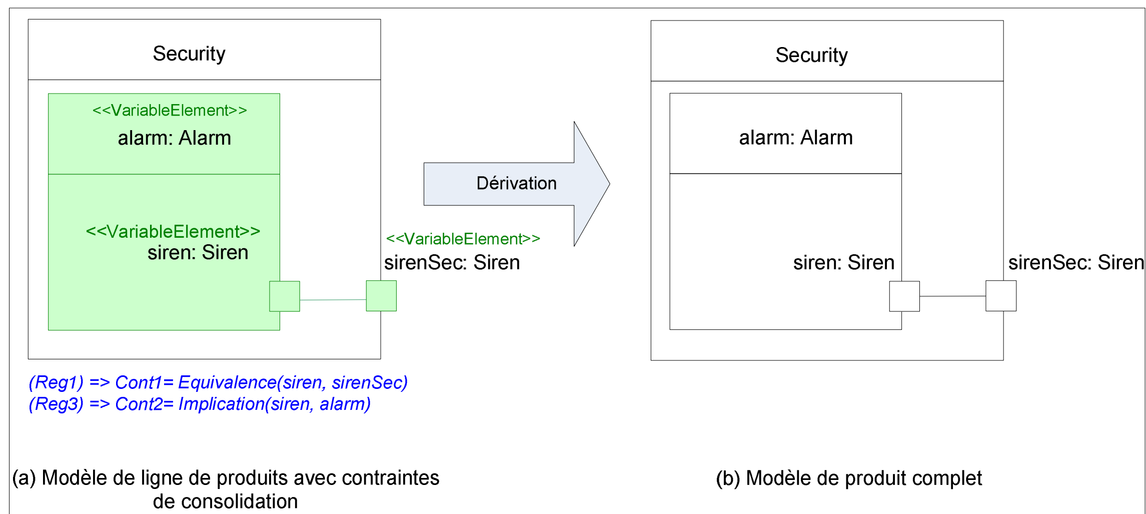


Figure6.11 : Après application des règles de consolidation *(Reg1)* et *(Reg3)*

6.4. Fusion de modèles : Evaluation et apports

Dans cette section, nous présentons quelques résultats d'évaluation de l'outil de fusion implémenté. Nous évaluons l'impact du nombre d'éléments communs sur le temps de la fusion.

6.4.1. Stratégie d'évaluation

Notre travail consiste à évaluer la capacité de passage à l'échelle de l'outil de fusion de modèles de lignes de produits logiciels. Pour ce faire, nous avons varié le nombre d'éléments des modèles à fusionner et nous avons mesuré le temps mis par l'outil pour les fusionner. De plus pour évaluer la manière avec laquelle le nombre d'éléments communs influence le temps de fusion, nous avons ajouté l'information qui représente le pourcentage d'éléments communs parmi les éléments à fusionner.

6.4.2. Résultats

Pour mener ce travail d'évaluation, nous avons choisi de mesurer le temps de fusion dans les cas où 50%, 70% et 100% des éléments à fusionner sont des éléments communs. Nous rappelons que les éléments communs sont des éléments correspondants selon les critères de correspondance que nous avons identifiés dans la section 4.4 du chapitre 4.

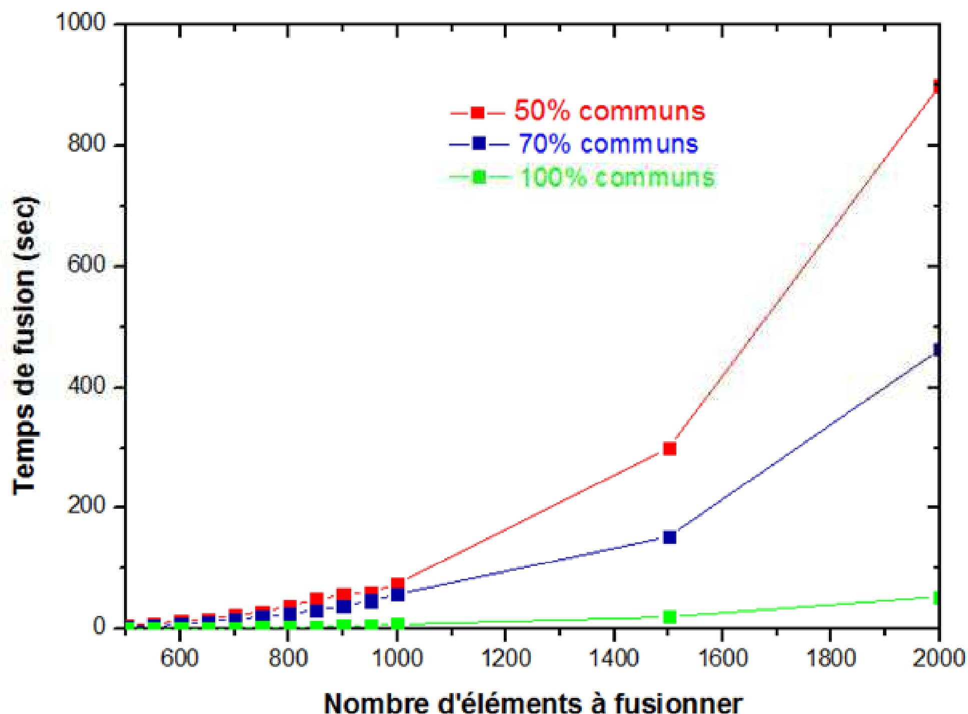


Figure 6.12 : Courbes du temps de fusion des modèles de lignes de produits logiciels

La figure 6.12 montre les résultats d'évaluation de l'outil de fusion. En effet, pour les trois pourcentages d'éléments en commun (50%, 70% et 100%) l'outil de fusion monte à l'échelle pour des temps de fusion raisonnables. Pour 2000 éléments à fusionner dont 50% sont des éléments communs, le temps moyen mis par l'outil de fusion atteint 900 secondes ce qui correspond à 15 minutes. Les valeurs de temps mesurées sont acceptables surtout pour des grands modèles de lignes de produits logiciels. La comparaison des courbes représentées dans la figure 6.12 montre que plus le nombre d'éléments en communs est élevé, plus le temps d'exécution mis par l'outil de fusion est court. Ces résultats peuvent être améliorés par optimisation des algorithmes implémentés dans l'outil de fusion. Cette amélioration consiste à écourter le temps mis par l'outil pour fusionner des modèles incluant de faibles pourcentages d'éléments communs.

6.5. Conclusion

Dans ce chapitre nous avons présenté l'environnement Sequoia dans lequel nous avons intégré les nouveaux outils. Nous avons ensuite donné quelques détails d'implémentation des outils de consolidation et de fusion des modèles de lignes de produits logiciels. Nous rappelons que l'outil de consolidation implémente les règles de consolidation précédemment mentionnées. L'outil de fusion, quant à lui, implémente les règles de fusion et les algorithmes

de fusion détaillés dans le chapitre 4. L'évaluation de l'outil de consolidation s'est intéressée à mesurer le nombre de produits incomplets éliminés. L'élimination de ces modèles incomplets permet au développeur de lignes de produits logiciels de gagner considérablement en termes de temps, d'effort et de coût. L'évaluation de l'outil de fusion a montré que ce dernier est capable de fusionner des modèles de lignes de produits logiciels larges et que le temps de fusion diminue avec l'augmentation du pourcentage d'éléments communs parmi les éléments des modèles de lignes de produits logiciels à fusionner. Les algorithmes implémentés par l'outil de fusion peuvent être optimisés pour réduire le temps de fusion. Les outils présentés et évalués dans ce chapitre sont utilisables dans plusieurs contextes. Par exemple pour l'outil d'évaluation, nous avons montré qu'il est indispensable pour la réutilisation des modèles de lignes de produits logiciels que cela soit dans le contexte d'une fusion, agrégation ou même pour l'utilisation basique dans le contexte de développement de lignes de produits logiciels. De même, l'outil de fusion peut être utilisable dans plusieurs contextes comme la fusion des fragments de modèles de lignes de produits logiciels développés par plusieurs intervenants, la fusion de différentes versions évoluées d'un modèle de ligne de produits logiciels et autres contextes d'utilisation.

CHAPITRE 7
Conclusion

Ce travail de thèse porte sur la gestion des lignes de produits logiciels complexes dans un contexte d'ingénierie dirigées par les modèles.

L'objectif est de traiter la problématique de composition des modèles de lignes de produits logiciels représentés sous formes de structures composites d'UML et incluant des annotations des éléments variables. Nous avons entamé nos travaux par une étude des approches de composition existantes. Nous nous sommes concentrés plus précisément sur les approches basées sur les modèles.

Notre étude des travaux de composition existants nous a permis d'identifier trois catégories d'approches.

La catégorie d'approches de composition basées sur les modèles de caractéristiques. L'utilisation des modèles de caractéristiques pour représenter les lignes de produits logiciels est une pratique répandue vu qu'elle permet de représenter les similarités et les différences sous une forme synthétique et concise. Toutefois, les mécanismes de composition proposés sont étroitement liés à la nature du modèle de caractéristiques. Ils ne peuvent pas être appliqués sur des modèles UML plus précisément sur l'aspect structurel qui est différent de celui des modèles de caractéristiques. La gestion de l'information de variabilité a été traitée explicitement pour certaines approches et gérée implicitement avec les contraintes de variabilité par d'autres approches. D'autre part, nous avons constaté que les mécanismes de composition des contraintes de variabilité restent dépendants des types de contraintes considérées.

La catégorie d'approches de composition basées sur la modélisation orientée aspects. Ces approches offrent l'avantage de modéliser les lignes de produits logiciels sous forme de modèles réutilisables nommés aspects. Leur principe de modélisation permet de gérer la variabilité d'une ligne de produits logiciels. En effet, pour obtenir un modèle de produit donné, il suffit de composer les aspects concernés avec le modèle de base. Ce principe de modélisation implique que le raisonnement reste local sur les aspects à composer. De ce fait, la variabilité n'est pas gérée au moment de la composition. Ce qui implique que les mécanismes adoptés par ces approches ne gèrent ni l'information de variabilité des éléments structurels ni les contraintes de variabilité durant la composition.

La catégorie d'approches de composition annotatrices de variabilité. Ces approches utilisent le langage de modélisation UML pour représenter les lignes de produits logiciels. Elles ont l'avantage de s'intéresser à la modélisation de la structure de la ligne de produits logiciels. De plus, elles permettent de distinguer les éléments structurels variables en utilisant des annotations spécifiques par le biais de stéréotypes UML. Cependant, ces approches ne proposent pas de mécanismes bien spécifiques pour la composition des modèles de lignes de produits logiciels incluant des éléments variables annotés. Dans le contexte d'utilisation du langage de modélisation UML, nous avons aussi étudié le mécanisme de fusion d'UML. C'est un mécanisme intéressant pour fusionner des modèles UML mais il ne traite pas la variabilité durant la fusion.

Pour remédier à de tels problèmes, nous avons proposé deux méthodologies de composition, la fusion et l'agrégation, qui considèrent la structure et l'information de variabilité comme points clés à gérer durant le processus de composition. Notre contribution a traité en premier temps l'aspect structurel des modèles de lignes de produits logiciels. Nous avons montré que sans consolidation des modèles de lignes de produits logiciels, les modèles de produits obtenus peuvent être structurellement incomplets et ne pourraient donc pas être fonctionnels. Produire des systèmes logiciels non fonctionnels implique des pertes majeures de temps, d'effort et de coût. Pour éviter ces problèmes, nous avons mis en œuvre un ensemble de règles de consolidation. Ces règles se matérialisent par la modification de l'information de variabilité de certains éléments structurels du modèle de ligne de produits logiciels ainsi que par l'ajout de contraintes structurelles. Les contraintes structurelles permettent de restreindre l'ensemble des modèles de produits obtenus à l'ensemble des modèles de produits structurellement complets. L'intérêt porté à la consolidation vient du fait qu'elle représente un traitement crucial à effectuer sur des modèles de lignes de produits logiciels en vue d'être utilisés ou réutilisés.

Nous avons ensuite enchainé avec les deux formes de composition proposées et nous les avons définies comme suit ; la fusion est un mécanisme de composition qui permet de combiner des modèles de lignes de produits logiciels ayant des parties similaires alors que l'agrégation est un mécanisme de composition qui permet de combiner des modèles de lignes de produits logiciels ne présentant pas de similitudes et pouvant être liés par des contraintes transversales.

Pour réaliser la fusion, nous avons mis en place un processus qui prend en entrée deux modèles de lignes de produits logiciels et qui se compose de trois phases. La première phase a pour objectif de consolider les modèles de lignes de produits logiciels en entrée. Les modèles

consolidés issues de cette phase sont ensuite fusionnés durant la deuxième phase, appelée phase de fusion. Cette phase propose des règles de fusion pour fusionner les éléments structurels ainsi qu'un algorithme générique pour combiner les contraintes de variabilité en respectant les propriétés sémantiques de la fusion. La troisième phase applique les règles de consolidation sur le modèle de ligne de produits résultant de la fusion.

L'étape suivante de la thèse s'est intéressée à l'agrégation. Pour ce faire, nous avons proposé un processus composé de deux phases. La première phase consolide les modèles en entrée. La deuxième phase présente deux alternatives d'agrégation dont la première oblige toutes les connexions entre les modèles à être présentes et la deuxième offre plus de liberté au concepteur de choisir les connexions qu'il veut faire apparaître entre les modèles.

Les contributions mises en œuvre dans cette thèse offrent la possibilité de composer des modèles de lignes de produits logiciels complexes. Les mécanismes de composition proposés s'intéressent aux structures composites des lignes de produits logiciels en considérant la variabilité comme un point clé à gérer durant la composition.

Cependant, cette dépendance aux structures composites d'UML et à l'aspect structurel des lignes de produits logiciels peut être considérée comme une limite. Toutefois, notre travail pourrait bien être étendu pour comprendre l'aspect comportemental des lignes de produits logiciels et utiliser d'autres modèles que les modèles UML.

Les perspectives de ce travail de recherche sont diverses. Nous pouvons penser à raffiner les mécanismes de composition pour qu'ils prennent en considération d'autres éléments UML plus détaillés (par exemple considérer les propriétés syntaxiques des méta-classes comme la propriété *isAbstract* pour la méta-classe *Class* durant la fusion), à étendre les mécanismes proposés pour qu'ils traitent l'aspect comportemental des modèles de lignes de produits logiciels, ou encore à considérer le problème de composition d'un point de vue plus général impliquant différents niveaux d'abstraction comme le niveau d'implémentation et le niveau exigences. Trouver des liens entre les modèles UML manipulés et des modèles plus synthétiques et compréhensibles, comme les modèles de caractéristiques par exemple, permettrait d'utiliser ces derniers pour faciliter la vérification des modèles UML à différentes phases du développement des lignes de produits logiciels surtout quand il s'agit de modèles complexes et à grande échelle.

Nous avons choisi de développer deux autres perspectives de ce travail à savoir l'amélioration du mécanisme de fusion pour qu'il prenne en compte la sémantique des éléments en plus de

leurs syntaxes et la proposition du mode intersection pour la fusion vu son utilité dans la détection d'évolution des modèles.

La sémantique pour l'amélioration des critères de fusion

La syntaxe est un des principaux critères à considérer pour fusionner des modèles UML. Cependant, des incohérences peuvent avoir lieu quand les modèles à fusionner sont développés par différents intervenants et provoquer ainsi une fusion défectueuse.
En effet, considérer uniquement la syntaxe comme critère de fusion pourrait mener à identifier comme éléments correspondants deux éléments structurels ayant la même signature mais représentant deux concepts différents. Prenons l'exemple d'un capteur de mouvement représenté dans un premier modèle par une classe nommée *Sensor*, et un autre capteur de température modélisé dans un deuxième modèle par une classe nommée *Sensor*. Dans ce cas, ayant des signatures équivalentes les deux classes correspondent et vont devoir être fusionnées bien qu'elles représentent des concepts différents.
D'autre part, la fusion peut ne pas avoir lieu quand il s'agit de deux éléments représentant un même concept mais possédant des signatures différentes. Prenons l'exemple d'un capteur de mouvement représenté dans un premier modèle par une classe *Sensor*, et un autre capteur de mouvement modélisé dans un deuxième modèle par une classe nommée *Detector*. Dans ce cas, ayant des signatures différentes les deux classes ne sont pas identifiées comme correspondantes et ne seront pas fusionnées bien qu'elles représentent le même concept qui est le capteur de mouvement.
France et al. [57] ont bien identifié ces problèmes pour fusionner des aspects. Ils ont proposé un ensemble de modifications que le concepteur peut faire manuellement avant et après la fusion. Cependant, cette solution ne garantit pas des omissions de la part du concepteur. Une solution automatisée doit être proposée pour détecter automatiquement quand deux concepts différents sont représentés par des éléments ayant le même nom ou quand un même concept est représenté par des éléments portant des noms différents. Des changements doivent être effectués automatiquement pour éviter les problèmes d'identification de concepts.
Prendre en compte la sémantique des éléments en plus de leur syntaxe lors de la fusion évite d'effectuer une fusion défectueuse.

La fusion en mode intersection

Les modèles de lignes de produits logiciels sont en évolution continue pour s'adapter aux besoins évolutifs des utilisateurs et suivre les progrès technologiques. Souvent quand un

modèle est développé en collaboration au sein d'une équipe, il subit plusieurs mises à jour. Ces mises à jour peuvent inclure l'évolution de certains éléments modélisés, l'émergence de nouveaux éléments ou aussi la modification d'éléments déjà existants dans le modèle. Pour identifier l'évolution entre deux versions du modèle de ligne de produits logiciels, la fusion en mode intersection représente une solution intéressante à réaliser. En effet, la fusion en mode intersection permet d'identifier les parties communes entre deux modèles de lignes de produits logiciels. Elle permet aussi de détecter le delta évolution entre deux versions de modèles de lignes de produits logiciels. Elle doit donc prendre en compte l'aspect structurel et/ou comportemental des modèles, l'information de variabilité des éléments à fusionner ainsi que les contraintes de variabilité.

Identifier le delta évolution permet aux concepteurs d'analyser par exemple l'impact de cette évolution sur les exigences initiales de la ligne de produits logiciels, sa conformité au profil utilisé ainsi que son impact sur les produits logiciels qui peuvent être ajoutés ou éliminés.

ANNEXE A

Expression de la variabilité dans SEQUOIA

- *Expression de la caractéristique optionnelle*

Dans un modèle de lignes de produits, certains éléments peuvent être optionnels. Un élément optionnel, dit aussi variable, est un élément du modèle qui peut être présent ou absent dans les modèles de produits d'une ligne de produits. Afin d'exprimer la variabililité des éléments, Sequoia définit le stéréotype « *VariableElement* » tel que décrit en Figure :

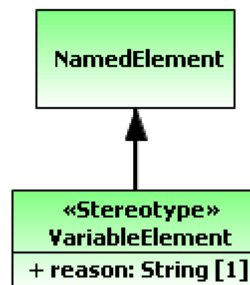


Figure A.1: Stéréotype pour l'expression de la variabilité

Afin d'exprimer la variabilité d'un élément, le concepteur l'annote en utilisant le stéréotype «*VariableElement* ». Il renseignera alors la propriété *reason* du stéréotype. Il s'agit d'un champ texte qui doit décrire la raison pour laquelle cet élément est optionnel.

- *Expression de la généricité*

Exprimer la généricité peut consister à paramétrer des éléments [63]. Par exemple dans un langage de programmation comme le C++, la notion de « *Template* » peut être utilisée pour la création de listes génériques. Ce concept existe également de manière native dans le langage UML. Cette notion peut être utilisée sur le concept de classe ou de paquetage comme illustré dans l'exemple de la figure A.2.

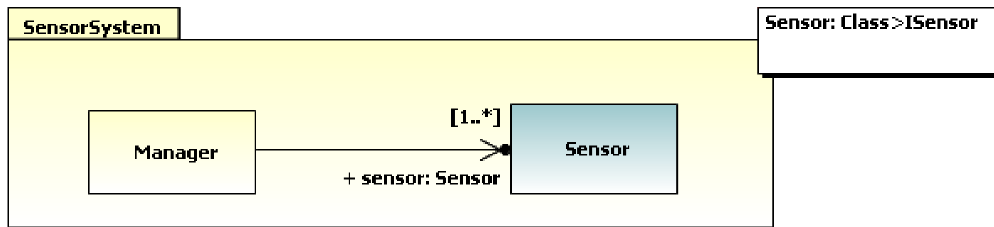


Figure A.2: Un système paramétré

L'exemple décrit dans la figure A.2 expose un gestionnaire associé à un capteur. Le paquetage *SensorSystem* contient deux classes, dont la classe *Sensor* est définie comme étant un paramètre. Ce paramètre possède une contrainte écrite de la manière suivante : *Sensor :Class>ISensor.* Cela indique que le paramètre doit au moins implanter l'interface ISensor.

Pour associer une valeur au paramètre, et obtenir un système sans paramètre, il faut effectuer une substitution de paramètres. Dans la Figure , le paramètre est substitué à la fois par la classe *TemperatureSensor* et *WindSensor*. Il est à noté que le langage UML permet la substitution d'un paramètre plusieurs fois, comme c'est le cas dans notre exemple (Figure).

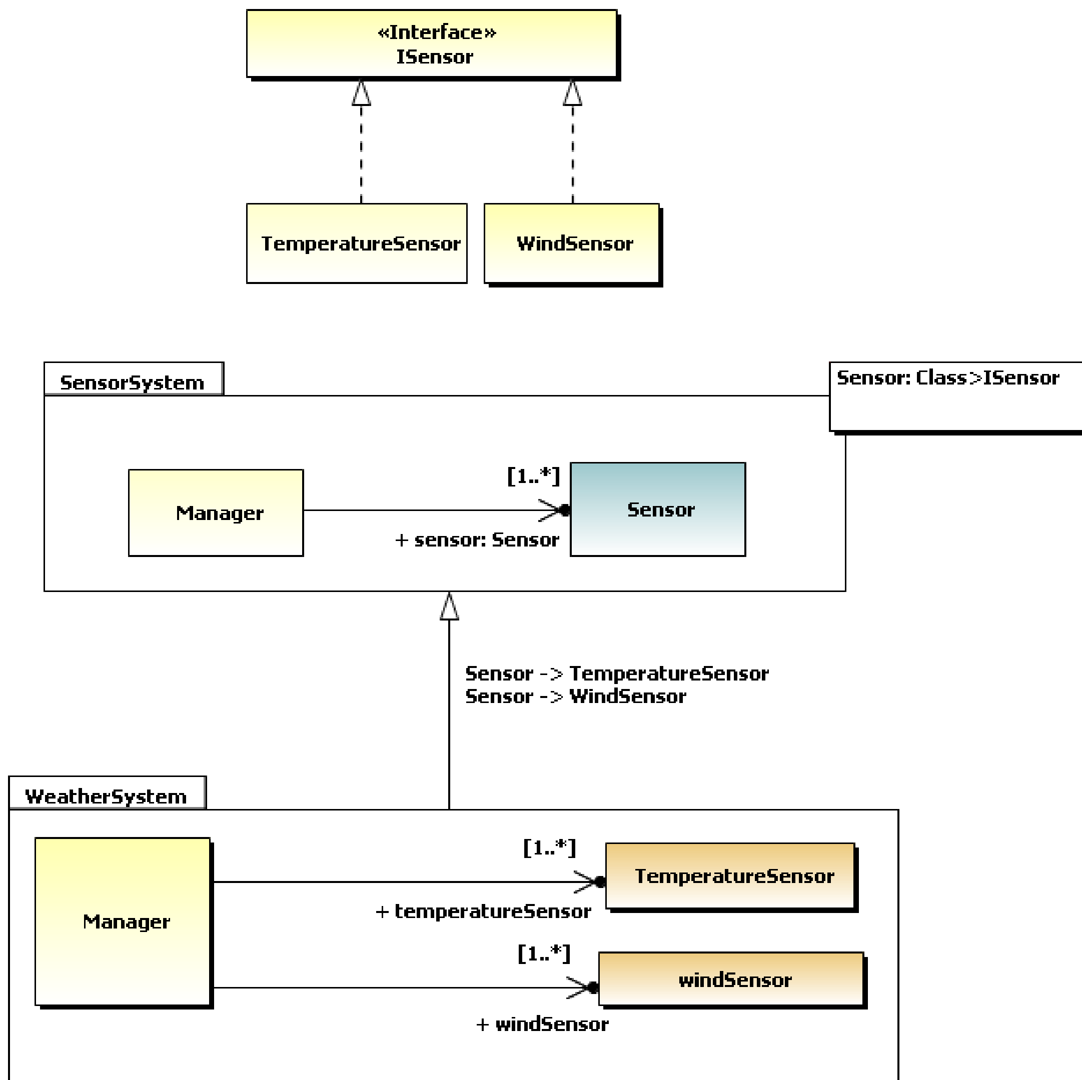


Figure A.3: Substitution du paramètre

Dans le cas où le concepteur ne souhaite autoriser qu'une seule substitution du paramètre, le profil Sequoia propose le stéréotype *« RestrictedTemplateParameter »* à appliquer sur l'élément *TemplateParameter* [2] (figure A.4). Ainsi le paramètre stéréotype ne peut être substitué qu'une seule fois.

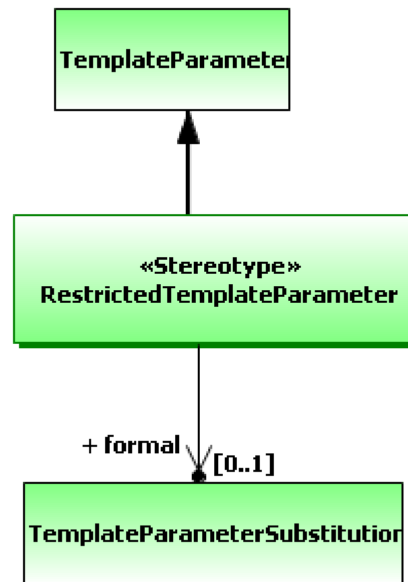


Figure A.4: Le stereotype ResctrictedTemplateParameter [2]

Le mécanisme de *template* du langage UML est très puissant. D'une part, il est possible d'ajouter un élément *Template* sur un paquetage ou sur une classe, et d'autre part le paramètre peut être tout élément nommé. Cela permet une grande expression pour la généricité dans un modèle.

Note : Le concept de Template et d'élément variable peuvent être utilisés sur le même élément. Cet ajout d'information signale que le paramètre est optionnel.

Ajout de contraintes

Exprimer le fait qu'un élément est variable ou paramétrable n'est pas suffisant. En fonction des besoins, il est nécessaire de contraindre les choix possibles pour les paramètres ou encore de contraindre la présence d'un élément variable. Afin d'exprimer ce type d'information, des contraintes peuvent être ajoutées. Elles permettent d'exprimer n'importe quelle expression logique entre des éléments variables ou des paramètres.

Pour ce faire, le profil séquoia (figure A.5) propose un ensemble de stéréotypes qui s'appliquent sur un élément de type contrainte.

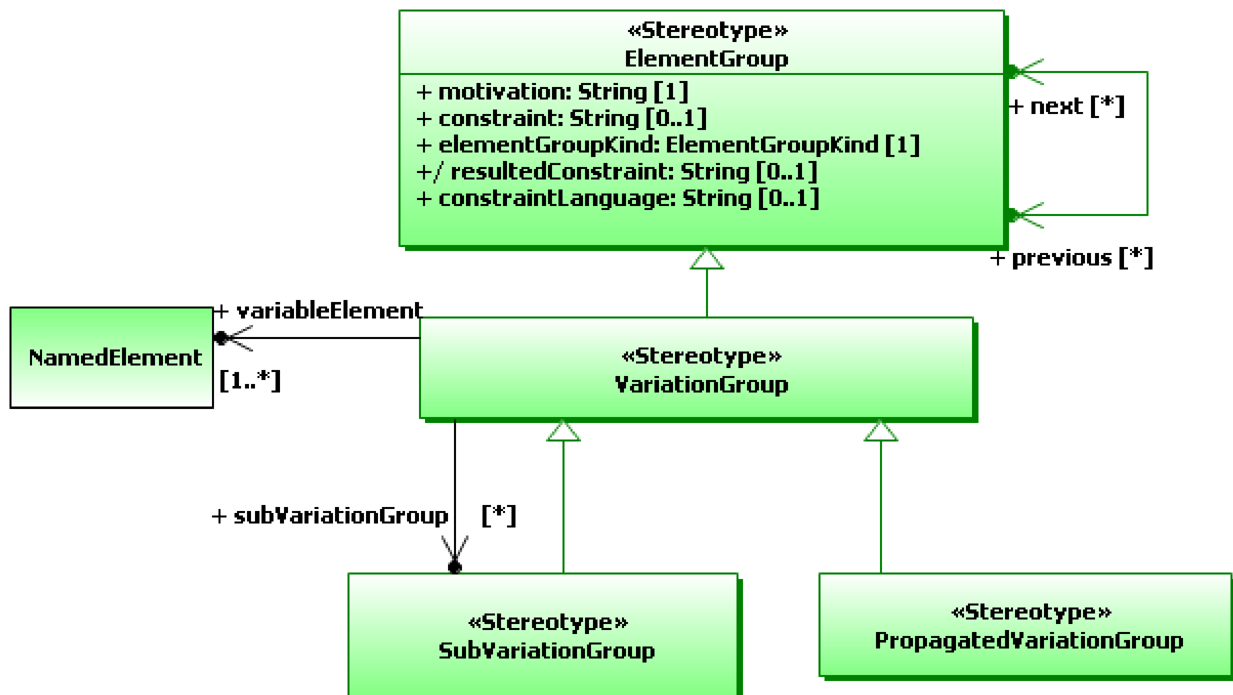


Figure A.5: Groupe de variations

«ElementGroup»

Tous les stéréotypes de type contrainte introduits dans Sequoia étendent le stéréotype abstrait *ElementGroup*. Ils héritent donc des propriétés suivantes :

motivation : cette propriété est une chaine de caractère. L'auteur de cette contrainte doit remplir cette chaîne afin d'expliquer pourquoi cette contrainte a été ajoutée.

elementGroupKind : pour cette propriété l'auteur indique la nature de la contrainte. Les différents choix possibles sont définis dans l'énumération *ElementGroupKind*

constraintLanguage : cette propriété définie le langage de contrainte utilisé pour les champs *constraint* et *resultedConstraint*.

constraint : En ayant choisi le type « *CustomizedCombination* » pour la propriété *elementGroupKind,* l'auteur écrit ici sa propre contrainte dans le langage spécifié par *ConstraintLanguage*. Ce champ est optionnel, il n'est rempli que si le type de contraintes spécifié dans *ElementGroupKind* est insuffisant pour le concepteur.

resultedConstraint : Ce champ peut être utilisé pour convertir la contrainte exprimée par le groupe dans un langage textuel de type *constraintLanguage.* Par exemple il est possible d'exprimer en OCL la contrainte décrite dans un groupe de variation.

«VariationGroup»

Le stéréotype « *VariationGroup* » expriment une relation logique entre les éléments variables ou les éléments qui peuvent être des paramètres. Pour cela, le stéréotype référence une liste qui contient soit des éléments variables ou des éléments qui peuvent être des valeurs possibles pour des paramètres. Cette liste est ordonnée. La contrainte s'obtient en appliquant l'operateur sur la liste ordonnée.

En fonction du type de contrainte choisie (voir figure A.6), voici la contrainte logique correspondante pour le l'exemple de la A.7.

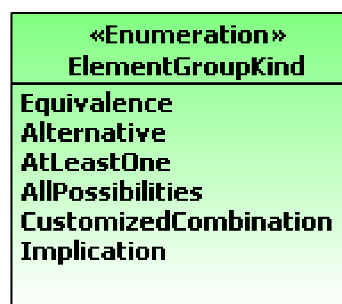


Figure A.6: Détails de l'énumération ElementGroupKind

Afin d'illustrer les différentes contraintes que l'on peut exprimer en Sequoia, les expressions logiques sont décrites à partir d'un exemple. La figure A.6 montre un modèle dans lequel il y a deux éléments variables a et b et un groupe de variation qui contraint ces deux éléments.

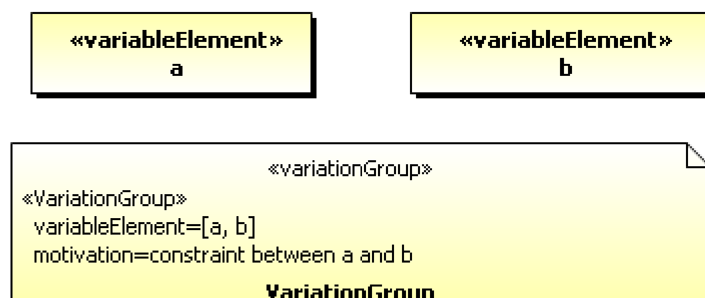


Figure A.7: Cas d'utilisation du groupe de variations

Implication

Dans le cas où *elementGroupKind* est égale à l'implication, l'expression logique associée à ce groupe de variation est $a \rightarrow b = \bar{a} + b$.

a	b	$a \rightarrow b$
false	**false**	**true**
false	**true**	**true**
true	**false**	**false**
true	**true**	**true**

Equivalence

Dans le cas où *elementGroupKind* est égale à *Equivalence*, l'expression logique associée à ce groupe de variation $a \leftrightarrow b = (\bar{a}.\bar{b}) + (a.b)$:

a	b	$a \leftrightarrow b$
false	**false**	**true**
false	**true**	**false**
true	**false**	**false**
true	**true**	**true**

AtLeastOne

Dans le cas où *elementGroupKind* est égale à *atLeastOne*, l'expression logique associée à ce groupe de variation $a + b$:

a	b	AtleastOne
false	**false**	**false**
false	**true**	**true**
true	**false**	**true**
true	**true**	**true**

Allposibilities

Dans le cas où *elementGroupKind* est égale à *allPossibilities*, l'expression logique associée à ce groupe de variation est :

a	b	AllPossibilities
false	**false**	**true**
false	**true**	**true**
true	**false**	**true**
true	**true**	**true**

Alternative

Dans le cas où *elementGroupKind* est égale à *alternative*, l'expression logique associée à ce groupe de variation $a \; xor \; b = \bar{a}b + a\bar{b}$:

A	B	$a \; xor \; b$
false	**false**	**false**
false	**true**	**true**
true	**false**	**false**
true	**true**	**true**

Note : Il est intéressant de noter que les groupes de variations référencent les éléments variables ou les paramètres. Les contraintes sont ainsi indépendantes du modèle de la ligne de système. On peut donc définir plusieurs ensembles de contraintes pour un même modèle de ligne de systèmes.

«SubVariationGroup»

Dans certain cas, certaines contraintes peuvent être plus complexes à réaliser. Le concepteur peut vouloir lier deux groupes de variations par un autre groupe de variation. Le calcul issu de cette structure est alors différent et nécessite de distinguer les groupes des sous-groupes de variations. Pour cette raison, Sequoia introduit le concept de sous-groupe de variations au travers du stéréotype « SubVarationGroup ».

Structuration d'un modèle de ligne de produits

Concevoir une ligne de produits est un investissement important et il est donc attendu que ce soit fait pour durer. Afin, de rendre cela possible, la structure de celle-ci doit être suffisamment flexible. Il faut que les différents éléments constituants de la ligne de produits ne soient pas monolithiques. Pour arriver à cet objectif, le profil Sequoia propose une structure générique du modèle de la ligne de systèmes [2] (figure A.8) :

Un modèle de ligne de produit stéréotypé « ProductLineModel » : Il s'agit du modèle de conception de la ligne de systèmes. Il contient tous les éléments du système, incluant les éléments variables. Il faut noter qu'un élément qui n'est pas variable est commun à tous les produits.

Un modèle stéréotypé « ConstraintConfigurationModel » contient toutes les contraintes, c'est-à-dire les groupes de variations référençants les variantes possibles, que ce soient les éléments de modèles annotés «VariableElement«, ou les paramètres de modèles template.

Un modèle stéréotypé « ConstraintConfigurationModelSet ». Celui-ci est le conteneur de modèle décrivant les différentes configurations possibles d'une ligne de produit. Ces derniers sont annotés du stéréotype « ConstraintConfigurationModel ». De cette manière il est possible, pour une ligne de systèmes, d'associer plusieurs ensemble de contraintes.

Un modèle stéréotypé « ProductModelSet » est un conteneur de « ProductsModel ». Celui-ci référence le modèle de ligne de systèmes.

Le « ProductsModel » va contenir l'ensemble des modèles de produits que l'on peut générer à partir du modèle de la ligne de systèmes et d'un ensemble de contraintes de type « ConstraintConfigurationModel ».

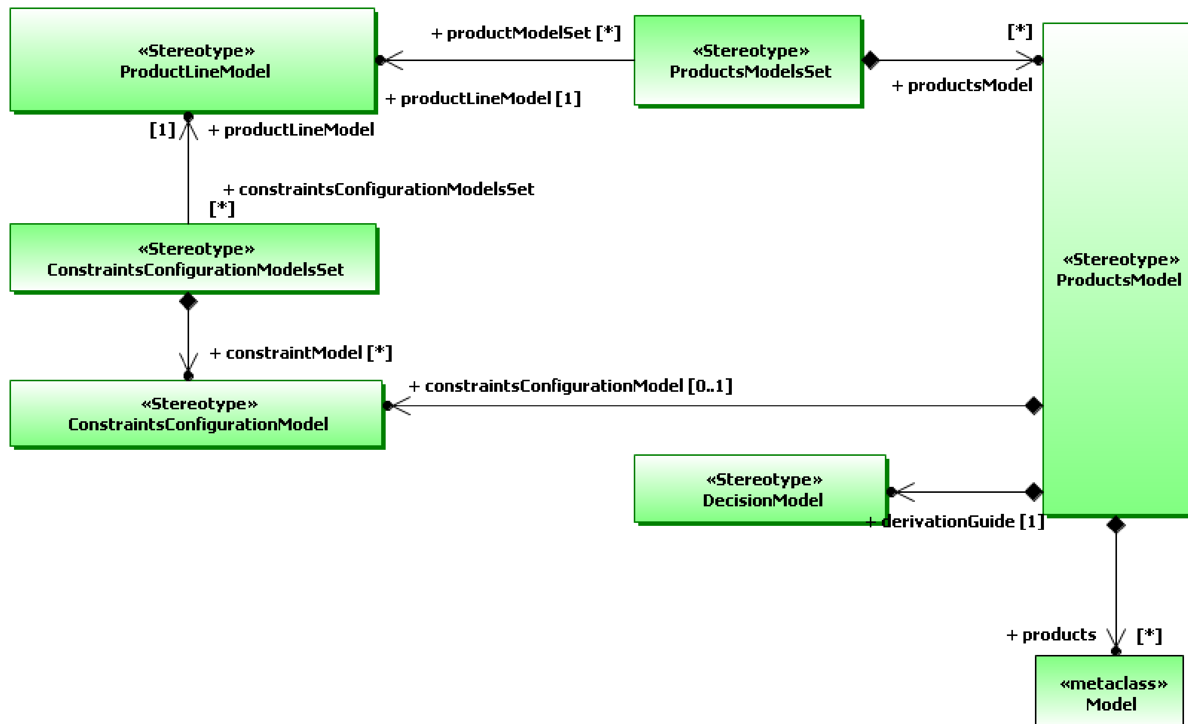


Figure A.8: Concepts lié à la structuration générique d'une ligne de systèmes avec Sequoia

Note : un modèle de produit issu de la ligne de produits référence cette dernière la réciproque n'est pas vraie. Ainsi si le modèle de produit est modifié pour diverses raisons, le modèle de la ligne de produits n'est pas impacté.

Bibliographie

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, et A. S. Peterson, « Feature-Oriented Domain Analysis (FODA) Feasibility Study », Technical report, CMU/SEI TR-21, USA, nov. 1990.

[2] K. Czarnecki et U. Eisenecker, *Generative Programming: Methods, Techniques and Applications*, 1er éd. USA: Addison Wesley, 2000.

[3] C. Atkinson, J. Bayer, et D. Muthig, « Component-Based Product Line Development: The KobrA Approach », in *First Product Line Conference (SPLC), Kluwer International Series in Software Engineering and Computer Science*, 2000, p. 19.

[4] H. Gomaa, « Designing Software Product Lines with UML », in *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop - Tutorial Notes*, Washington, DC, USA, 2005, p. 160–216.

[5] D. Dhungana, P. Grünbacher, R. Rabiser, et T. Neumayer, « Structuring the modeling space and supporting evolution in software product line engineering », in The Journal of Systems and Software, vol. 83, no. 7, p. 1108-1122, juill. 2010.

[6] D. L. Parnas, « On the criteria to be used in decomposing systems into modules », *Communication of the ACM*, vol. 15, no. 12, p. 1053–1058, déc. 1972.

[7] A. K. Thurimella, B. Bruegge, et O. Creighton, « Identifying and Exploiting the Similarities between Rationale Management and Variability Management », in *Proceedings of the 2008 12th International Software Product Line Conference*, Washington, DC, USA, 2008, p. 99–108.

[8] A. Helferich, K. Schmid, et G. Herzwurm, « Product management for software product lines: an unsolved problem? », *Communication of the ACM*, vol. 49, no. 12, p. 66–67, déc. 2006.

[9] K. C. Kang, P. Donohoe, E. Koh, J. Lee, et K. Lee, « Using a Marketing and Product Plan as a Key Driver for Product Line Asset Development », in *Proceedings of the Second International Conference on Software Product Lines*, London, UK, UK, 2002, p. 366–382.

[10] A. Jaaksi, « Developing Mobile Browsers in a Product Line », *in Journal of IEEE Software*, vol. 19, no. 4, p. 73–80, juilly. 2002.

[11] « UML 2.4.1 »http://www.omg.org/spec/UML/2.4.1/.

[12] D. M. Weiss et C. T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley Professional, 1999.

[13] P. Clements et L. Northrop, *Software Product Lines: Practices and Patterns*, 1er éd. Addison Wesley, 2001.

[14] K. Pohl, G. Böckle, et F. V. D. Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Birkhäuser, 2005.

[15] C. W. Krueger, « Software reuse », *ACM Computing Surveys*, vol. 24, no. 2, p. 131–183, juin 1992.

[16] D. Mcilroy, « Mass-produced Software Components », in *Proceedings of Software Engineering Concepts and Techniques*, Garmisch, Germany, 1969, p. 138-155.

[17] D. L. Parnas, « On the Design and Development of Program Families », *in the Journal of IEEE Transactions on Software Engineering*, vol. 2 , issue no. 1, p. 1 - 9, mars 1976.

[18] R. L. Glass, « Frequently forgotten fundamental facts about software engineering », *in the Journal of IEEE Software, IEEE*, vol. 18, no. 3, p. 112 -111, mai 2001.

[19] S. Davis, «*Future Perfect*» *10th Anniversary Edition*, Addison-Wesley Pub Co, Harlow, England, 1996.

[20] J. Kramer, « Is abstraction the key to computing? », *in the Communications of the ACM*, vol. 50, no. 4, p. 36–42, avr. 2007.

[21] I. Jena et M. Clauß, *Generic Modeling using UML extensions for variability*. In *Proceedings of OOPSLA Workshop on Domain-specific Visual Languages* (2001), pp. 11-18.

[22] T. Ziadi, L. Hélouët, et J.-M. Jézéquel, « Towards a UML Profile for Software Product Lines », in *Software Product-Family Engineering*, vol. 3014, F. J. Linden, Edition Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, p. 129-139.

[23] Ø. Haugen, B. Møller-pedersen, J. Oldevik, et A. Solberg, « An MDA-based framework for model-driven product derivation », *in the SOFTWARE ENGINEERING AND APPLICATIONS*, p. 709-714, 2004.

[24] M. Clauß et I. Jena, « Modeling variability with UML », in *In GCSE 2001Young Researchers Workshop*, 2001.

[25] P. Tessier, «*Conception de modèles de familles de systèmes temps réel*». 2005.

[26] P. Zave et M. Jackson, «*Four Dark Corners Of Requirements Engineering*», i*n the Journal of ACM Transactions on Software Engineering and Methodology (TOSEM)*, Volume 6 Issue 1, Jan.1997 Pages1-30, USA 1997.

[27] K. C. Kang, S. Kim, J. Lee, K. Kim, G. J. Kim, et E. Shin, « FORM: A feature-oriented reuse method with domain-specific reference architectures », *in the Journal of Annals of Software Engineering*, vol. 5, p. 143–168, 1998.

[28] P. J. Bosch, «*Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*», 1er edition. Addison Wesley, 2000.

[29] D. Batory, J. Liu, et J. N. Sarvela, « Refinements and multi-dimensional separation of concerns », in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2003, p. 48–57.

[30] D. Batory, « Feature models, grammars, and propositional formulas », *in SPLC'05 Proceedings of the 9th international conference on Software Product Lines*, Pages 7-20 Springer-Verlag Berlin, Heidelberg, 2005.

[31] K. Chen, W. Zhang, H. Zhao, et H. Mei, « An approach to constructing feature models based on requirements clustering », in *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, 2005, p. 31 - 40.

[32] A. Classen, P. Heymans, et P.-Y. Schobbens, « What's in a feature: a requirements engineering perspective », in *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, Berlin, Heidelberg, 2008, p. 16–30.

[33] S. Apel et C. Kästner, «*An Overview of Feature-Oriented Software Development*», *in the Journal of Object Technology*, vol.8, p.49-84,jully 2009.

[34] M. L. Griss, J. Favaro, et M. d'Alessandro, « Integrating Feature Modeling with the RSEB », in *Proceedings of the 5th International Conference on Software Reuse*, Washington, DC, USA, 1998, p. 76–.

[35] M. L. Griss, « Implementing Product-Line Features with Component Reuse », in *Proceedings of the 6th International Conerence on Software Reuse: Advances in Software Reusability*, London, UK, UK, 2000, p. 137–152.

[36] M. L. Griss, «*Product-Line Architectures*» in *Component-Based Software Engineering*, G. T. Heineman and W. Councill, Eds.: Addison-Wesley, 2001, p.405-419.

[37] M. Eriksson, J. Börstler, et K. Borg, « The PLUSS Approach – Domain Modeling with Features, Use Cases and Use Case Realizations », in *Software Product Lines*, vol. 3714, H. Obbink et K. Pohl, Éd. Springer Berlin / Heidelberg, 2005, p. 33-44.

[38] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, et C. Lucena, « Refactoring product lines », in *Proceedings of the 5th international conference on Generative programming and component engineering*, New York, NY, USA, 2006, p. 201–210.

[39] S. Segura, D. Benavides, A. Ruiz-cortés, et P. Trinidad, « Automated Merging of Feature Models using Graph Transformations » in Generative and Transformational Techniques in Software Engineering II, Berlin, 2008,p. 489-505.

[40] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, et Y. Bontemps, « Generic semantics of feature diagrams », *in the International Journal of Computer and Telecommunication Networking*, vol. 51, n°. 2, p. 456–479, USA, 2007.

[41] P. Heymans, P.-Y. Schobbens, J.-C. Trigaux, Y. Bontemps, R. Matulevičius, et A. Classen, « Evaluating formal properties of feature diagram languages », *in the IEEE Proceedings Software*, vol. 2, n°. 3, p. 281, 2008.

[42] M. Acher, P. Collet, P. Lahire, et R. France, « Composing Feature Models », *in Software Language Engineering*, vol. 5969, M. Brand, D. Gašević, et J. Gray, Edition Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 62-81.

[43] van den broek, « Merging Feature Models », *In 14th International Software Product Line Conference*, SPLC 2010, v. 2, p.83-89, 2010, Jeju Island, South Korea.

[44] S. Apel, F. J, S. Trujillo, et C. Kästner, « Model Superimposition in Software Product Lines », in *Proceedings of the International Conference on Model Transformation (ICMT), volume 5563 of LNCS*, p. 4-19, 2009.

[45] S. Apel et C. Lengauer, « Superimposition: A Language-Independent Approach to Software Composition », in *Software Composition*, 2008, p. 20-35.

[46] K. Czarnecki, C. H. Peter Kim, et K. T. Kalleberg, « Feature Models are Views on Ontologies », in *Proceedings of the 10th International on Software Product Line Conference*, Washington, DC, USA, 2006, p. 41–51.

[47] « Aspect Oriented Modeling Workshop ». http://www.aspect-modeling.org/.

[48] K. Alfert, « Requirements, Features and Aspectsfor Software Product Lines », in *Aspects and Software Product Lines: An Early Aspects Workshop at SPLC-Europe 2005*, 2005.

[49] G. Chastek et J. D. McGregor, *« Early Aspects in Software Product Line in Product Production », in the 6th Workshop on Early Aspects*, France, 2005.

[50] A. Nyßen, S. Tyszberowicz, et T. Weiler, « Are Aspects useful for Managing Variability in Software Product Lines? A Case Study », in *In Aspects and Software Product Lines: An Early Aspects Workshop at SPLC*, 2005.

[51] H. Siy, M. Z, et V. Winter, «*The Role of Aspects in Domain Engineering*», *In Aspects and Software Product Lines (An Early Aspects Workshop at SPLC-Europe),* 2005.

[52] I. Groher et M. Voelter, « XWeave: models and aspects in concert », in *Proceedings of the 10th international workshop on Aspect-oriented modeling*, New York, NY, USA, 2007, p. 35–40.

[53] I. Groher et M. Voelter, « Expressing Feature-Based Variability in Structural Models », in *Workshop on Managing Variability for Software Product Lines*, 2007.

[54] Iris Groher et M. Voelter, « Using Aspects to Model Product Line Variability », *in Workshop on Early Aspects: Aspect-Oriented Requirements and Architecture for Product Lines*, SPLC, Limerick, Ireland, pp. 89-95, 2008.

[55] R. Reddy et R. France, « Model Composition - A Signature-Based Approach », in Aspect Oriented Modeling(AOM) Workshop, Montego, 2005.

[56] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. Mceachen, E. Song, et G. Georg, « Directives for composing aspect-oriented design class models », *In the Journal Transactions on Aspect-Oriented Software Development I*, Vol. 3880, Springer (2006) , p. 75-105.

[57] R. France, F. Fleurey, R. Reddy, B. Baudry, et S. Ghosh, « Providing Support for Model Composition in Metamodels », in *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, 2007, p. 253-253.

[58] S. Clarke et R. J. Walker, « Composition patterns: an approach to designing reusable aspects », in *Proceedings of the 23rd International Conference on Software Engineering*, Washington, DC, USA, 2001, p. 5–14.

[59] S. Clarke, « Extending standard UML with model composition semantics », *in The Journal of Science of Computer Programming-special issue on Unified Modeling Language (UML)* , vol. 44, n°. 1, p. 71–100, juill. 2002.

[60] M. Acher, P. Collet, P. Lahire, et R. France, « Comparing Approaches to Implement Feature Model Composition », in *Modelling Foundations and Applications*, vol. 6138, Edition Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 3-19.

[61] T. Ben Rhouma, P. Tessier, F. Terrier, « Merging UML2 composite structures of Software Product Lines », *in proceedings of the ICECCS 2012, 17th IEEE International Conference on Engineering of Complex Computer Systems,* p.77-85.

[62] J. P. Marques-silva et K. A. Sakallah, « GRASP: A Search Algorithm for Propositional Satisfiability », *IEEE Transactions on Computers*, vol. 48, p. 506–521, 1999.

[63] A. Cuccuru, A. Radermacher, S. Gérard, et F. Terrier, *« Constraining Type Parameters of UML 2 Templates with Substitutable Classifiers »,* in *Model Driven Engineering Languages and Systems*, vol. 5795, Edition Springer Berlin / Heidelberg, 2009, p. 644-649.