Configuration par modèle de caractéristiques adapté au contexte pour les lignes de produits logiciels

Thibaut Possompès

ACADÉMIE DE MONTPELLIER

UNIVERSITÉ **M**ONTPELLIER **II**

Sciences et **T**echniques du **L**anguedoc

THÈSE

présentée au Laboratoire d'Informatique de Robotique
et de Microélectronique de Montpellier pour
obtenir le diplôme de doctorat

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

Configuration par modèle de caractéristiques adapté au contexte pour les lignes de produits logiciels

Application aux *Smart Buildings*

par

Thibaut POSSOMPÈS

Soutenance prévue pour le 4 octobre 2013, devant le jury composé de :

Rapporteurs
Laurence DUCHIEN, Professeur ..... LIFL, Université Lille 1
Jean-Paul RIGAULT, Professeur ..... I3S, Université de Nice Sophia Antipolis

Examinateurs
François BRIANT, Distinguished Engineer ..... IBM France, PSSC Montpellier
Bernard COULETTE, Professeur ..... IRIT, Université Toulouse II

Invités
Anne LAURENT, Professeur ..... LIRMM, Université Montpellier II
Xavier VASQUES, IBM Manager ..... IBM France, PSSC Montpellier

Co-Encadrants de thèse
Christophe DONY, Professeur ..... LIRMM, Université Montpellier II
Marianne HUCHARD, Professeur ..... LIRMM, Université Montpellier II
Chouki TIBERMACINE, Maître de conférences ..... LIRMM, Université Montpellier II

Table des matières

Table des matières **i**

1 Introduction **1**

2 Le projet RIDER, la modélisation et l'optimisation énergétique de bâtiments **5**
2.1 Cas d'étude – Le projet RIDER 5
2.2 Méthodes d'optimisation énergétique 8
2.3 Architecture logicielle pour l'optimisation énergétique de bâtiments 10
2.3.1 Description des architectures logicielles typiques 10
2.3.2 Architecture logicielle de RIDER 13
2.4 La représentation des bâtiments 14
2.4.1 Usages d'un modèle de bâtiments 15
2.4.2 Modèles de bâtiments existants 16
2.4.3 Un nouveau méta-modèle de bâtiments pensé pour l'optimisation énergétique 18
2.5 Conclusion 25

3 Introduction aux lignes de produits logiciels et modèles de caractéristiques **27**
3.1 Les lignes de produits logiciels et leur cycle de vie 28
3.2 Les différents types de variabilité 29
3.3 Modèles de caractéristiques 32
3.3.1 Définition 33
3.3.2 Les usages des modèles de caractéristiques 34
3.3.3 Adaptation de modèles de caractéristiques 36
3.4 Conclusion 37

4 Un nouveau méta-modèle de caractéristiques pour la prise en compte automatisée du contexte, et son implémentation **39**
4.1 Introduction 39

4.2 Synthèse des concepts décrits par les méta-modèles de caractéristiques existants . . . 40
4.2.1 Définition du concept de *caractéristique*, ou *feature* . . . 41
4.2.2 Relations entre caractéristiques . . . 42
4.2.3 Groupements arbitraires de caractéristiques . . . 47
4.2.4 Informations liées aux produits et aux configurations . . . 48
4.3 Méta-modèle proposé . . . 48
4.4 Outillage du méta-modèle proposé . . . 55
4.4.1 Profil UML . . . 56
4.4.2 La navigabilité des classes du modèle métier . . . 58
4.4.3 Outillage . . . 61
4.5 Comparaison avec les outils existants . . . 65
4.6 Conclusion . . . 65

5 Génération automatisée de modèles de caractéristiques dépendants du contexte par transformation de modèles 67
5.1 Introduction . . . 67
5.2 Description globale de l'approche . . . 68
5.3 Définitions et notations . . . 69
5.4 Algorithme de génération du modèle de caractéristiques spécifique au contexte 88
5.5 Implémentation . . . 92
5.6 Conclusion . . . 93

6 Application du méta-modèle et des algorithmes proposés dans le contexte du projet RIDER 97
6.1 Introduction . . . 97
6.2 Modèle de caractéristiques générique de RIDER . . . 99
6.3 Modélisation du bâtiment . . . 110
6.4 Modèle de caractéristiques adapté à un bâtiment . . . 115
6.5 Conclusion . . . 118

7 Conclusion et perspectives 119

A Profil UML 125
A.1 Présentation EBNF du nouveau méta-modèle de caractéristiques . . . 125
A.1.1 Forme EBNF . . . 125
A.1.2 Diagrammes syntaxiques . . . 127

CHAPITRE

1 Introduction

Ce travail se situe dans le contexte de la réalisation de logiciels pour le domaine des *bâtiments intelligents*, ou *smart buildings*. Ce domaine vise à instrumenter les bâtiments afin d'améliorer la gestion, par exemple, de leur consommation énergétique ou des équipements (maintenance des centrales des traitements d'air, occupation des salles, ...). Il s'agit d'une extension du domaine de la domotique [1] et d'une mise à l'échelle permettant de gérer des bâtiments ou des complexes industriels [Kolokotsa *et al.*, 2002; Brad Brech *et al.*, 2011].

Le projet RIDER ("Réseau et Inter connectivité Des Énergies classiques et Renouvelables") [2], financé par le programme FUI [3] pour le pôle de compétitivité DERBI [4], cherche à valoriser l'ensemble des instruments présents dans les bâtiments et à développer de nouvelles méthodes d'optimisation énergétique. Il regroupe un consortium d'entreprises et laboratoires de recherche, dont notamment IBM et le LIRMM, intéressés par l'amélioration de l'efficacité énergétique de bâtiments à l'aide des technologies de l'information. De plus, le projet a initié quatre thèses, dont celle-ci, concernant :

- la création de modèles physiques permettant l'optimisation énergétique,
- l'optimisation énergétique basée sur une modélisation du confort,
- la visualisation interactive en 3D de bâtiments,
- l'adaptation d'un logiciel au contexte de chaque bâtiment par le biais d'un modèle de caractéristiques.

Le consortium a notamment pour objectif commun de lever les verrous scientifiques et techniques suivants :

1. La domotique permet le contrôle des équipements d'un bâtiment, mais ce terme est généralement employé dans le cadre d'applications domestiques.
2. http://rider-project.com/
3. Fonds unique interministériel – http://competitivite.gouv.fr/
4. http://www.pole-derbi.com

- Modéliser les environnements concernés par l'optimisation énergétique, sur les plans énergétique et informatique,
- Gérer des volumes de données importants issus de nombreux capteurs,
- Utiliser en temps réel des algorithmes de fouille de données et d'aide à la décision pour économiser de l'énergie,
- Mettre en œuvre des réseaux de capteurs sur de grandes distances à moindre coût,
- Coordonner l'utilisation de différentes sources énergétiques,
- Permettre le déploiement à grande échelle de logiciels d'optimisation énergétique,
- Analyser l'impact social de la mise en œuvre d'approches permettant l'économie d'énergies.

La problématique générale que cette thèse adresse, concerne le déploiement à grande échelle d'un logiciel d'optimisation énergétique. Cela consiste à déterminer comment adapter un logiciel, pour lequel il est possible de réutiliser ou paramétrer différents composants existants, à son contexte d'exécution. Par exemple, les fonctionnalités d'un logiciel de gestion d'une entreprise peut être adapté en fonction des besoins spécifiques de chaque département et de son organigramme. Dans le cas du projet RIDER, nous souhaitons pouvoir adapter un logiciel d'optimisation énergétique en fonction, par exemple, des protocoles et interfaces des instruments présents dans chaque pièce, ou selon les optimisations possibles dans chaque zone d'un bâtiment.

Nous nous sommes appuyés sur l'approche par *lignes de produits logiciels* [Klaus Pohl *et al.*, 2005] pour adapter les composants logiciels du projet RIDER à chaque bâtiment. Cette approche repose sur l'utilisation d'un *modèle de caractéristiques*, aussi appelé *feature model* [Kang *et al.*, 1990]. Son rôle est d'organiser hiérarchiquement des éléments appelés *caractéristiques*, ou *features*, qui décrivent l'ensemble des fonctionnalités, paramétrages, ou propriétés d'un ensemble de composants logiciels réutilisables. Les contraintes de compatibilité ou d'incompatibilité entre les caractéristiques sont aussi représentées dans ce modèle. Un modèle de caractéristiques représente l'ensemble des configurations possibles permettant de générer un logiciel fonctionnel à partir des composants logiciels disponibles.

Dans la littérature, il existe un grand nombre de modèles de "modèles de caractéristiques" (méta-modèles de caractéristiques) [Trigaux et Heymans, 2003]. Chacun répond à des besoins différents. Par exemple, le méta-modèle de Fey *et al.* [2002] a pour objectif la simplification du traitement algorithmique d'un modèle de caractéristiques. Cependant, son édition manuelle peut se révéler fastidieuse à cause du grand nombre de relations à représenter. Au contraire, les méta-modèles utilisés par Kang *et al.* [1990] et Kang *et al.* [1998] visent d'abord à simplifier la documentation des caractéristiques d'un domaine ainsi que leurs contraintes d'interdépendance pour être utilisé comme un moyen de communication entre les personnes impliquées dans un projet de développement logiciel. Pour parvenir à modéliser efficacement les caractéristiques de la ligne de produits RIDER, nous avons choisi de créer un nouveau méta-modèle synthétisant les principaux méta-modèles existants.

La principale spécificité du projet RIDER est que chaque produit issu de la ligne de produits doit être adapté aux spécificités du bâtiment dont il doit optimiser l'énergie. Nous n'avons pas trouvé de solution satisfaisante à cette problématique. Des approches répondant à des problématiques similaires ont été proposées [Acher *et al.*, 2009; Fernandes *et al.*,

2011], mais celles-ci ne répondent pas aux besoins particuliers du projet RIDER. Nous souhaitons pouvoir déterminer sur quels éléments du contexte vont avoir un impact les caractéristiques du futur produit.

Nos contributions portent sur les deux sujets suivants :

- Nous avons cherché les concepts majeurs utilisés pour décrire les caractéristiques de logiciels parmi les approches existantes par lignes de produits. Ainsi, nous proposons un nouveau méta-modèle synthétisant des méta-modèles de caractéristiques existants au regard des contraintes du projet dans lequel se déroule cette thèse [Thibaut Possompès *et al.*, 2010a; Xavier Vasques *et al.*, 2011]. De plus, nous avons ajouté de nouveaux concepts permettant d'associer des caractéristiques logicielles à des éléments du contexte qui ont un impact sur leur présence. Nous avons mis en œuvre notre nouveau méta-modèle dans une approche par ingénierie des modèles basée sur le langage UML [Thibaut Possompès *et al.*, 2010b]. Nous avons choisi d'implémenter notre méta-modèle sous la forme d'un profil UML sur lequel nous avons basé les outils développés durant cette thèse [Thibaut Possompès *et al.*, 2011a].
- Nous avons proposé la spécification d'une méthodologie d'adaptation automatique de modèle de caractéristiques à un contexte d'exécution [Thibaut Possompès *et al.*, 2011b]. Notre spécification définit les modèles nécessaires (contexte et caractéristiques) et l'algorithme d'adaptation. Notre méthode consiste à identifier les caractéristiques dont la présence dans un produit dépend de la présence de certains éléments du contexte. L'algorithme génère un modèle de caractéristiques spécifique au contexte dans lequel chaque caractéristique se rapportant à un concept du contexte est associée à un élément, instance du concept, auquel elle se rapporte [Thibaut Possompès *et al.*, 2013].

Notre mémoire sera composé de sept chapitres. Nous présentons un cas d'étude de logiciel adapté au contexte dans le chapitre 2. Il s'agit d'un logiciel d'optimisation énergétique de bâtiments en cours de réalisation dans le cadre du projet RIDER.

Le chapitre 3 donne une vision d'ensemble du domaine des lignes de produits logiciels et présente les travaux de recherche autour de ce sujet (les modèles de caractéristiques et l'impact que le contexte d'exécution du produit peut avoir sur ses caractéristiques).

Le chapitre 4 présente notre première contribution. Après un état de l'art concernant les modèles de caractéristiques, nous en faisons la synthèse en proposant un nouveau méta-modèle. Nous proposons également des nouveautés concernant la liaison entre caractéristiques et contexte. La contribution relative à ce chapitre a fait l'objet d'une intégration dans le logiciel de modélisation *IBM Rational Software Architect* (RSA).

Le chapitre 5 propose une méthode d'adaptation de modèles de caractéristiques au contexte. Nous formalisons les concepts clefs de notre nouveau méta-modèle, puis nous présentons l'algorithme de transformation de modèles de caractéristiques. La contribution relative à ce chapitre a été intégrée et testée au sein d'un prototype.

Le chapitre 6 montre un exemple d'application de nos contributions au projet RIDER.

Ce chapitre décrit le modèle de caractéristiques d'un logiciel d'optimisation énergétique en utilisant l'approche développée dans les chapitres précédents.

Le chapitre 7 conclut cette thèse. Il en présente une synthèse et récapitule notamment nos contributions et développe des perspectives pour de nouveaux travaux.

CHAPITRE

2 Le projet RIDER, la modélisation et l'optimisation énergétique de bâtiments

Sommaire

2.1 Cas d'étude – Le projet RIDER 5
2.2 Méthodes d'optimisation énergétique 8
2.3 Architecture logicielle pour l'optimisation énergétique de bâtiments 10
2.4 La représentation des bâtiments 14
2.5 Conclusion 25

Ce chapitre présente une vue d'ensemble du contexte et des problématiques du projet dans lequel se situe cette thèse. Nous proposons plus particulièrement un aperçu global du domaine de l'optimisation énergétique de bâtiments. Cet aperçu englobe les principales méthodes d'optimisation énergétique, les architectures logicielles couramment utilisées pour instrumenter et optimiser un bâtiment, et les modèles de données de bâtiments.

La section 2.1 présente et introduit le cadre du projet dans lequel se place cette thèse. La section 2.2 introduit différentes méthodes d'optimisation énergétique applicables dans le contexte du projet. La section 2.3 décrit des architectures logicielles existantes permettant l'optimisation énergétique de bâtiment à l'aide des méthodes précédemment présentées. La section 2.3.2 présente une synthèse de l'architecture logicielle retenue pour le logiciel d'optimisation énergétique de bâtiments. La section 2.4 décrit différents modèles de bâtiments existants. La section 2.4.3 présente le nouveau modèle de bâtiments que nous avons réalisé pour décrire les bâtiments selon nos besoins. La section 2.5 conclut ce chapitre et introduit les besoins liés à cette nouvelle architecture.

2.1 Cas d'étude – Le projet RIDER

A l'heure actuelle, la gestion énergétique de bâtiments est effectuée par des systèmes de gestion technique centralisée (GTC). Une GTC est un logiciel auquel sont connectés les

capteurs et actionneurs, et de façon générale, tous les équipements d'un ou plusieurs bâtiments. Son rôle consiste à recevoir toutes les données envoyées par les capteurs et à envoyer des ordres aux actionneurs selon des règles définies et les données reçues. Une GTC peut superviser :

- les systèmes de ventilation, chauffage, air conditionné,
- les alarmes,
- les lumières,
- les contrôles d'accès,
- la distribution d'eau, de gaz.

C'est un logiciel temps réel, très fiable, basé sur un matériel de faible capacité. Il permet de réguler la consommation énergétique selon des règles préétablies, mais ne peut le faire de façon optimale car il n'a pas la puissance de calcul nécessaire. Les occupants du bâtiment utilisent la GTC pour avoir une vision globale de la situation du bâtiment, l'état des équipements, et spécifier les consignes de température.

La figure 2.1 montre un extrait de plan de bâtiment dont les salles sont équipées de capteurs de température et d'hygrométrie, et d'actionneurs de chauffage. Tous ces instruments sont reliés à une GTC centrale, par un réseau filaire.

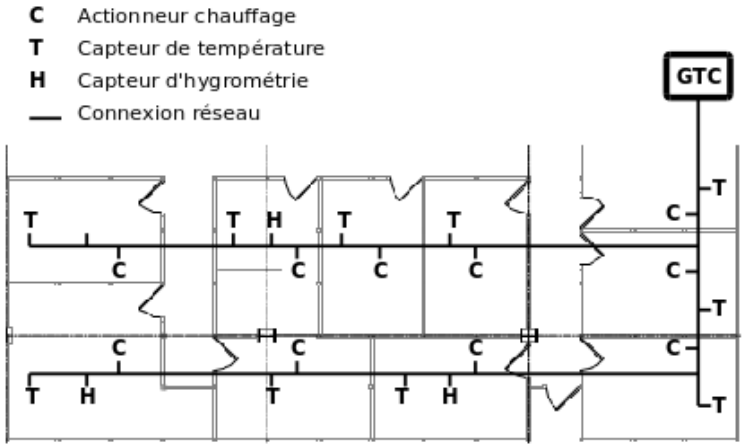

FIGURE 2.1 : Usage d'une GTC

Le projet RIDER [Xavier Vasques *et al.*, 2011] vise à améliorer l'efficacité énergétique de bâtiments en apportant une surcouche logicielle à cet environnement pour effectuer des analyses et recommander à la GTC un fonctionnement optimal. Notre problématique consiste à déterminer comment réutiliser, avec des bâtiments différents, les logiciels réalisés pour les sites pilotes du projet.

Un système RIDER est associé aux GTC d'un bâtiment, ou d'un groupe de bâtiments. Il dispose des interfaces de communication nécessaires pour communiquer avec chaque GTC. Cela peut nécessiter l'usage de protocoles et de formats de données différents. De plus, le système peut communiquer avec des services supplémentaires afin d'améliorer la précision des prédictions énergétiques. Par exemple, l'utilisation d'un service de prévisions météorologiques, et d'un service d'agenda d'occupation des salles.

La figure 2.2 schématise l'utilisation du système d'optimisation énergétique. Le cadre central représente le système RIDER. Son fonctionnement interne sera décrit dans les sections suivantes. Le système montré dans la figure communique avec deux GTC afin d'optimiser l'énergie de différents bâtiments. Chaque GTC gère les capteurs et actionneurs du bâtiment dans lequel elle est installée.

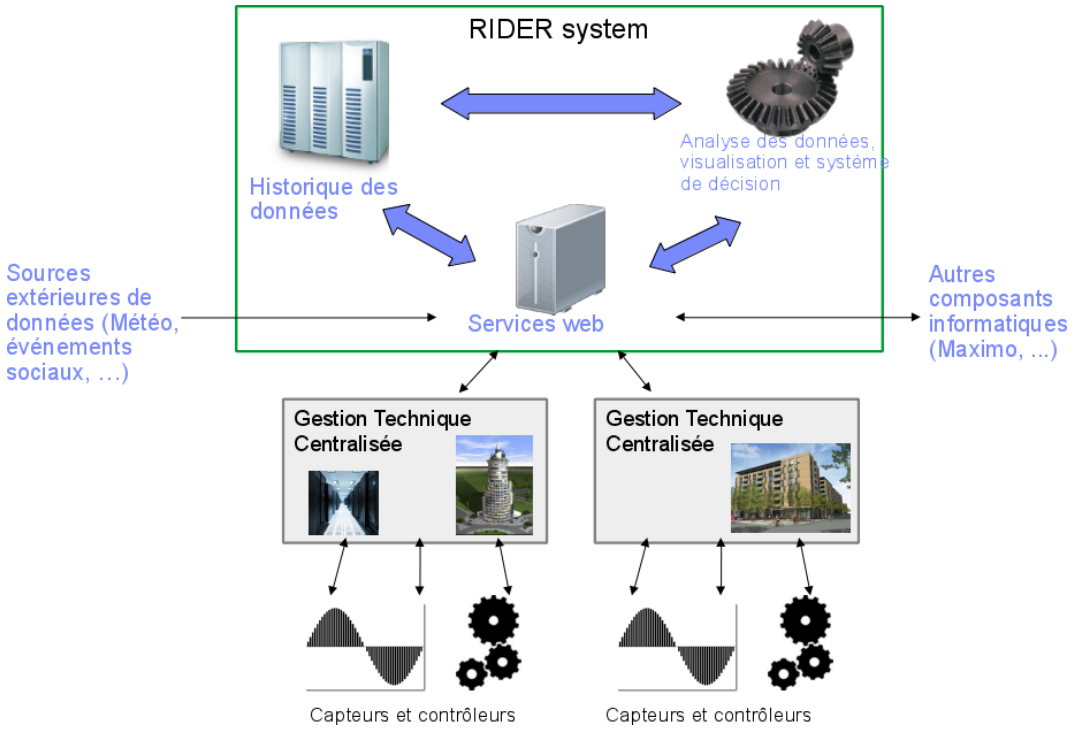

FIGURE 2.2 : Vue d'ensemble de RIDER

Les résultats d'analyses et prédictions du système peuvent être consultés sur des tableaux de bords destinés aux habitants et aux gestionnaires des bâtiments. Concrètement, l'optimisation énergétique est faite en proposant des valeurs de consignes aux GTC en temps réel (par exemple, température, recyclage de l'air) ou de façon planifiée (par exemple, une température optimisée pour les salles réservées). La GTC possède le dernier mot sur la prise en compte ou non de la consigne pour des raisons de sécurité et réglementaires.

Ainsi, pour un bâtiment donné, il faut déterminer la disponibilité des caractéristiques du logiciel d'optimisation en analysant le modèle du bâtiment selon des critères qui sont spécifiques à chaque fonctionnalité du logiciel.

Pour répondre à cette problématique, nous proposons de modéliser un logiciel d'optimisation énergétique de bâtiment, et ses méthodes d'optimisation, sous la forme d'une ligne de produits logiciels. Nous décrivons, de façon théorique, dans les chapitres 3, 4 et 5 les détails de notre approche par lignes de produits logiciels. Dans le chapitre 6 nous montrons une application de nos contributions dans le contexte de l'optimisation énergétique de bâtiments.

Dans la suite de ce chapitre, nous allons examiner comment fonctionnent des approches d'optimisation énergétique similaires.

2.2 Méthodes d'optimisation énergétique

La facture énergétique d'un bâtiment peut être réduite de deux façons :

1. en ayant recours à des mesures d'efficacité énergétique passive,
2. en améliorant l'efficacité énergétique de façon active.

La première solution consiste à améliorer par exemple l'isolation du bâtiment, à choisir des équipements plus performants (par exemple, fenêtres, stores) [Pröglhöf *et al.*, 2011]. Cette conception peut être réalisée par exemple à l'aide de patrons architecturaux [Gholipour *et al.*, 2009]. L'efficacité énergétique d'un bâtiment peut également être améliorée en réaménageant son environnement à l'aide, par exemple, de végétation [Akbari, 2002].

Apporter une amélioration énergétique de façon active consiste à mieux exploiter les équipements et les flux énergétiques du bâtiment. Cet objectif peut être atteint en appliquant par exemple les méthodes suivantes :

- adapter le débit d'air soufflé par la ventilation en fonction de l'occupation de la salle ou du niveau de CO_2,
- doser la quantité d'air repris à l'extérieur pour diminuer le coût du chauffage,
- changer les consignes de chauffage en fonction du planning d'occupation.

Un dernier facteur entrant en œuvre dans l'efficacité énergétique est l'implication des habitants. S'ils cherchent à participer activement aux économies d'énergies, une part importante des économies sera effectuée seulement grâce à leur comportement et à la modification de leurs habitudes. Différentes méthodes permettent d'impliquer les habitants (par exemple, jeux, compétitions, récompenses), toutes nécessitent la présence d'interfaces utilisateurs. Il existe différents travaux visant à réaliser des interfaces utilisateurs appropriées pour visualiser [Benoit Lange, 2012] ou contrôler [Kolokotsa *et al.*, 2002; Dounis et Caraiscos, 2009] des systèmes d'optimisation énergétique ou des GTC. Penner et Steinmetz [2002] constatent que les interfaces utilisateur de GTC nécessitent un lourd travail d'adaptation et ne sont généralement pas totalement adaptées aux besoins des utilisateurs. Ils proposent une approche par ingénierie des modèles pour générer automatiquement les interfaces utilisateurs.

Nous ne rentrons pas dans les détails de ces méthodes d'optimisation car cela sort du cadre de cette étude. Mais nous en faisons un tour d'horizon afin de connaître les principales méthodes de travail du domaine de l'optimisation énergétique de bâtiments, et quels types de méthodes d'optimisation pourront être réutilisés dans de nouveaux bâtiments.

L'article *Advanced control systems engineering for energy and comfort management in a building environment – A review* [Dounis et Caraiscos, 2009] détaille les différentes approches permettant l'optimisation énergétique de bâtiments. Chaque approche (par exemple, réseau de neurones, systèmes multi-agents, ...) permet d'optimiser des paramètres précis (par exemple, CO_2, luminosité, consommation énergétique, contrôle de la température, de la ventilation) qui entrent en jeu dans la consommation énergétique. Et de plus, chaque approche dispose de caractéristiques précises qui peuvent présenter des avantages ou désavantages selon l'usage du bâtiment (contrôle du confort thermique, stratégie d'optimisation globale au bâtiment, donner la priorité aux techniques d'optimisation passives, respecter les préférences des utilisateurs, adaptation automatique, apprentissage).

Des heuristiques [Huang et Lam, 1997; Kalogirou, 2000, 2001] permettent de prédire et de calculer des données manquantes, et de contrôler des équipements notamment dans les domaines suivants :

- prédiction de la production d'une centrale électrique solaire,
- modélisation et prédiction de la production d'un chauffe-eau solaire,
- prédiction de l'énergie nécessaire pour chauffer un bâtiment,
- calcul du renouvellement de l'air dans une pièce,
- prédiction de l'ensoleillement et de la force du vent,
- prédiction du coût de l'énergie.

L'approche proposée par Livengood et Larson [2009] vise à proposer un petit ordinateur connecté aux équipements électriques d'un bâtiment dont le rôle est de déterminer en temps réel comment utiliser le plus efficacement l'électricité (par exemple, revente, stockage, activation d'un équipement très consommateur). Leur objectif est de prendre en compte la variation du prix de l'électricité et de la demande pour décider algorithmiquement d'un usage optimal de l'énergie. Daniel Kullmann et Henrik W. Bindner [2011] proposent une approche répondant à des problématiques similaires au niveau du réseau électrique qui interconnecte plusieurs bâtiments.

La thèse de Manuel Bauer [1998] développe des modèles thermiques et météorologiques pour mieux gérer les contrôleurs de stores et de chauffage d'un bâtiment. Il cherche également à restreindre le nombre de capteurs nécessaires à ses modèles. Il atteint jusqu'à 33% d'économies d'énergies.

Dans le cadre du projet RIDER, les méthodes d'optimisation énergétique développées seront de la nature suivante :

- *Optimisation du système de chauffage et de climatisation en fonction de l'occupation réelle du bâtiment.* Cela consiste à calculer le moment optimal où lancer le chauffage ou la climatisation. Les résultats dépendent de l'inertie thermique du bâtiment et de son occupation.
- *Optimisation du système de ventilation, chauffage, et air conditionné, compte tenu du confort des habitants.* Cela consiste à doser le débit d'air, la température de l'air soufflé et la quantité d'air repris à l'extérieur afin d'adapter la concentration de CO2, l'humidité et la température pour que les occupants du bâtiment aient la meilleure sensation de confort à moindre coût.
- *Détection de gaspillage énergétique.* Le logiciel peut identifier des situations où l'énergie est gaspillée, par exemple, lumières allumées dans une pièces vide, et prévenir les habitants ou agir sur les équipements.

La section suivante présente des architectures logicielles utilisées généralement dans le cadre de logiciels d'optimisation énergétique.

2.3 Architecture logicielle pour l'optimisation énergétique de bâtiments

Cette section présente les fonctionnalités clefs offertes par une architecture logicielle permettant l'optimisation énergétique d'un ou plusieurs bâtiments. L'optimisation énergétique est une fonctionnalité de haut niveau. Cependant, elle est fortement tributaire des couches logicielles et matérielles inférieures lui permettant d'accéder aux données et de contrôler indirectement un bâtiment. Ainsi, dans cette section, nous aborderons des architectures qui peuvent être plus ou moins liées à la GTC et aux instruments du bâtiment. Par exemple, Kwang-Soon Choi *et al.* [2011] présentent la mise en œuvre de capteurs et contrôleurs dans une habitation, et leur liaison réseau avec un serveur central. Cet article détaille les fonctionnalités des capteurs (par exemple, communication, mode veille), les fonctionnalités du serveur (par exemple, protocoles de communication, monitoring de la consommation d'électricité). Leur approche consiste à analyser les consommations d'électricité des équipements présents dans une habitation, et à couper l'électricité du mode veille. L'architecture qu'ils utilisent peut être réutilisée pour mettre en œuvre d'autres méthodes d'optimisation.

2.3.1 Description des architectures logicielles typiques

Il existe un grand nombre d'architectures différentes utilisables dans le domaine de l'optimisation énergétique de bâtiments. Nous ne pouvons toutes les énumérer mais nous décrivons dans cette sous section les architectures les plus représentatives de celles utilisées, ou susceptibles d'être utilisées, dans l'industrie :

- L'architecture *Sensor Andrew* [Rowe *et al.*, 2011] vise à la gestion des capteurs et actionneurs de parc immobiliers étendus. Les auteurs cherchent à uniformiser l'accès aux capteurs et aux actionneurs indifféremment de leur zone ou de leurs protocoles de communication. Ainsi, leur approche permet à des applications d'optimisation énergétique d'accéder à un ensemble de zones instrumentées qui seraient autrement isolées les unes des autres. Cette architecture a été réalisée pour optimiser l'énergie en analysant des patterns de consommation et pour identifier les gaspillages. C'est une architecture trois tiers composée des couches suivantes :

  1. *Couche des transducteurs* : Cette couche est constituée de capteurs et d'actionneurs (aussi appelés instruments ou transducteurs) dont le rôle est d'échanger des informations et des commandes avec la couche des passerelles.
  2. *Couche des passerelles* : Cette couche permet de faire le pont entre les instruments de la couche des transducteurs et internet. Elle est constituée de périphériques connectés à internet. Ils convertissent les données des instruments, décrites dans un format bas-niveau, vers un format ouvert adapté aux serveurs.
  3. *Couche des agents* : Cette couche contient les serveurs, les applications clients (appelées *agents*), et les nœuds dédiés au traitement des événements issus de la couche des passerelles.

- L'architecture suivante [R. Zach et A. Mahdavi, 2010; R. Zach *et al.*, 2011] permet la mesure simultanée de la consommation énergétique, de la performance, et de l'occupation de plusieurs bâtiments. Ils souhaitent permettre la prise en charge d'instruments de mesure existants, d'étendre une infrastructure de monitoring existante, et évaluer l'efficacité des modèles d'optimisation. Ils proposent une architecture générique en quatre niveaux :
  1. Le niveau *physique* contient tous les instruments de mesure et de contrôle du bâtiment.
  2. Le niveau *bus de communication* a pour but de transmettre les informations issues des instruments vers les automates. Il utilise des protocoles de communication spécialement adaptés aux instruments.
  3. Le niveau *automation* joue un rôle de passerelle. Il transfère toutes les données des capteurs vers le niveau management en utilisant un protocole de communication adapté au niveau management.
  4. Le niveau *management* est responsable des tâches de stockage, visualisation, et des traitements de données.
- Brad Brech *et al.* [2011] proposent une architecture pour gérer plus efficacement et éviter le gaspillage énergétique. Ils ont identifié six niveaux distincts :
  1. Le niveau *physique* contient les objets et les équipements du bâtiment qui doivent être mesurés.
  2. Le niveau *instrumentation* contient les GTC qui communiquent en temps réel avec les capteurs et actionneurs du bâtiment.
  3. Le niveau *interconnexion* agrège des données issues des différentes GTC et de différents bâtiments et normalise l'ensemble des données.
  4. Le niveau *intelligence* contient tous les outils d'analyse et de gestion du bâtiment.
  5. Le niveau *visualisation* permet la visualisation de l'ensemble des indicateurs de performance correspondant à un ou plusieurs bâtiments sur un tableau de bord. Cela permet la comparaison entre l'efficacité énergétique de plusieurs bâtiments.
- Une architecture avec modèle décisionnel à base de règles [Daniel Kullmann *et al.*, 2010; Daniel Kullmann et Henrik W. Bindner, 2011] peut être utilisée pour contrôler un ensemble de capteurs et d'actionneurs répartis sur un ensemble de sites différents. Les règles décrivent le comportement que doit adopter le système dans différentes situations. Par exemple, les règles décrivent sous quelles conditions le chargement de la batterie d'un véhicule électrique peut commencer (par exemple, coût de l'électricité, planning de charge, niveau de charge minimal requis pour effectuer le prochain déplacement).

L'implication et l'écoute des habitants du bâtiment optimisé est déterminante pour la réussite de la mise en œuvre d'un logiciel d'optimisation énergétique [Ali Keçebaş et İsmail Yabanova, 2010; Mohamad El Achkar, 2010]. Il existe de nombreux modèles permettant de prédire le comportement des habitants et de calculer leur niveau de confort. Malgré cela, il est utile de permettre au logiciel de réagir directement à des événements pour adapter son comportement de façon plus fine. Toutes les informations relatives au bâtiment, aux

habitants, aux événements ayant lieu dans le bâtiment, ou de façon générale, toutes les informations ayant un impact sur la consommation énergétique d'un bâtiment font partie du *contexte* dans lequel s'exécute un logiciel d'optimisation énergétique. Les architectures suivantes sont capables de prendre en compte de événements et les informations relatives au bâtiment pour adapter leur fonctionnement en fonction d'objectif préétablis :

- L'architecture logicielle nécessaire pour optimiser l'énergie d'un bâtiment s'apparente à celle des systèmes sensibles au contexte [Abowd *et al.*, 1999; Pascoe *et al.*, 1999]. Ce type de système doit être capable d'agréger des données issues d'un *contexte* (ici, le bâtiment) et de les traiter pour offrir des services appropriés (ici, des recommandations énergétiques, des consignes adaptées, des alertes pour les habitants) [Edwards et Grinter, 2001].
- Un système sensible au contexte utilise un "context broker" pour collecter des données et maintenir à jour une représentation du contexte. Un context broker est basé sur un moteur d'inférence afin de déduire de nouvelles informations [Jaroucheh *et al.*, 2009; Lee *et al.*, 2011; Lancia, 2008].
- Ce type de système a donné lieu à la création d'approches permettant de créer automatiquement le code d'un logiciel adapté à un contexte. Par exemple, Munoz *et al.* [2006] proposent une approche permettant de générer, par le biais d'une approche d'ingénierie dirigée par les modèles, une application sensible au contexte pour la gestion d'une salle de réunion. Cette application permet d'automatiser la gestion de la lumière (par exemple, en réunion, en utilisant ou non un vidéoprojecteur), et de surveiller les accès à la salle en dehors des périodes autorisées.
- De façon similaire, Pham *et al.* [2007] montrent différents usages d'ingénierie des modèles permettant la génération de logiciels embarqués. Les objectifs de ces approches sont :
  - la génération de code spécifique à différents périphériques à partir de modèles afin de réduire les besoins de maintenance,
  - la réalisation de frameworks permettant la mise en œuvre de logiciels sensibles au contexte (par exemple, des réseaux de capteurs et une architecture logicielle permettant de centraliser les informations),
  - faire abstraction de la complexité liée à l'hétérogénéité des périphériques et du matériel en utilisant des modèles de haut niveau.

Les principales architectures logicielles d'applications sensibles au contexte sont les suivantes [Baldauf *et al.*, 2007; Lee *et al.*, 2011] :

- Harry Chen et Anupam Joshi [2004] proposent l'architecture CoBrA dans laquelle le contexte est modélisé par une ontologie,
- Hofer *et al.* [2003] proposent l'architecture Hydrogen dans laquelle le contexte est modélisé à l'aide d'un méta-modèle générique.
- Brad Brech *et al.* [2011] proposent une architecture dans laquelle un context broker est utilisé pour traiter de façon appropriée les événements et alertes issus des GTC.
- Kiani *et al.* [2010] proposent une architecture qui peut être étendue à grande échelle en fédérant plusieurs context brokers. Les auteurs utilisent le langage ContextML [Strang et Linnhoff-Popien, 2004] pour représenter le contexte.

2.3.2 Architecture logicielle de RIDER

Les différentes propositions introduites dans la section précédente présentent des similarités concernant l'architecture de base nécessaire pour instrumenter un bâtiment, enregistrer les mesures et envoyer des ordres aux actionneurs. Nous retrouvons :

- une couche d'instrumentation contenant des capteurs et actionneurs reliés en réseau à l'aide de protocoles bas-niveau pour l'échange des données avec une couche de passerelles.
- une couche de passerelles contenant les GTC du bâtiment. Leur rôle est d'interagir avec les instruments via des protocoles de bas-niveau, de stocker ces données et de les transmettre à des serveurs par internet et via des protocoles ouverts.
- une couche de traitements, contenant des serveurs décentralisés dont le rôle est d'appliquer des algorithmes d'optimisation sur les données reçues et de renvoyer des recommandations vers la couche de passerelles. Cette couche peut également contenir des serveurs permettant l'affichage de tableaux de contrôle et des webservices afin d'agréger des données issues d'autres sources (par exemple, des prévisions météorologiques, des plannings).
  En résumé, cette couche est constituée des composants suivants :
  – une interface de communication pour GTC,
  – d'interfaces avec des services extérieurs,
  – d'un data warehouse,
  – de composants d'optimisation énergétique,
  – d'interfaces utilisateur de contrôle.

La mise en œuvre d'une méthode d'optimisation plutôt qu'une autre ne change pas l'organisation de l'architecture logicielle précédemment décrite. Dans le cadre de l'optimisation de plusieurs bâtiments (supervision à grande échelle), nous avons vu que des mécanismes spécifiques sont utiles pour simplifier la gestion d'un grand nombre de GTC, déterminer une politique d'utilisation des ressources énergétiques, permettre la mutualisation de logiciels d'optimisation, et la standardisation des accès aux GTC et aux instruments pour faire abstraction de la diversité des systèmes et protocoles de communication.

À cela doivent s'ajouter des composants logiciels permettant l'optimisation énergétique. Si plusieurs méthodes d'optimisation sont utilisées pour une même zone, un context broker peut permettre la définition de règles pour déterminer quelles recommandations prendre en compte ou écarter. Le context broker peut aussi être utilisé pour spécifier des actions à effectuer lorsque certaines situations se présentent, déterminer quelles sources énergétiques favoriser, choisir une "politique" d'économie d'énergie, déduire de nouvelles informations à partir des données reçues.

Compte tenu du contexte industriel et des travaux existants, nous nous trouvons dans une situation où il s'agit d'une architecture à grosse granularité intégrant des logiciels développés par des tiers. Nous n'avons pas la possibilité de générer ces logiciels, mais nous pouvons adapter leur configuration en fonction des informations du bâtiment. Par exemple, nous pouvons spécifier quelles zones doivent être traitées par un module d'optimisation

énergétique qui utilise le retour des utilisateurs pour affiner ses résultats. Contrairement aux cas traités par les approches fondées sur l'ingénierie des modèles présentées par [Pham *et al.*, 2007; Munoz *et al.*, 2006], nous nous trouvons dans un contexte dans lequel le code des applications ne peut pas être généré car il n'est pas disponible.

Cependant, il existe une grande quantité de méthodes d'optimisation qui sont dépendantes de l'instrumentation des bâtiments et des attentes des utilisateurs. Le choix d'une méthode appropriée aux différentes zones d'un bâtiment (par exemple, bureaux, salle machine, entrepôt, salle de réunion, salon, chambre) dépend des instruments disponibles, des objectifs économiques, et des profils des habitants (par exemple, habitant d'un logement, salarié, gestionnaire de bâtiments industriels).

L'architecture choisie par le consortium pour RIDER est décrite dans la figure 2.3. Elle décrit de façon synthétique l'organisation des principaux composants logiciels de notre architecture. Plusieurs GTC peuvent être utilisés pour superviser un ou plusieurs bâtiments. La partie dans le cadre en pointillés verts décrit les composants dont le nombre et l'organisation est spécifique à chaque bâtiment (par exemple, chaque bâtiment est instrumenté différemment et possède des GTC différentes). Le composant *Agrégation de données* permet de faire abstraction du nombre et du type de GTC utilisées dans le bâtiment. Il sert à recevoir les données et envoyer des consignes aux GTC. Les données reçues sont chargées dans le data warehouse qui est utilisé pour alimenter les modèles d'optimisation. En retour, les modules d'optimisation envoient des consignes aux GTC par l'intermédiaire du composant d'agrégation.

Comme dit précédemment, il s'agit d'une architecture à grosse granularité. Nous considérons les modules d'optimisation comme des composants non décomposables, et dont le paramétrage est limité. Si nécessaire, l'un des modules d'optimisation peut jouer le rôle de context broker afin de résoudre des conflits de consigne ou déduire de nouvelles informations à partir des données du data warehouse.

Les modules d'optimisation énergétique se basent sur les données issues du data warehouse. Nous avons en effet choisi d'utiliser un modèle central pour représenter l'intégralité des informations relatives aux bâtiments. Les informations disponibles concernant un bâtiment déterminent quels modules d'optimisation peuvent être utilisé. La section suivante décrit les différentes méthodes permettant de modéliser un bâtiment.

2.4 La représentation des bâtiments

Tout système d'optimisation énergétique de bâtiments a besoin d'un modèle de données permettant de représenter les informations liées au bâtiment. Cette section décrit les différents usages d'un modèle de bâtiments et présente les principales caractéristiques des modèles de bâtiments existants.

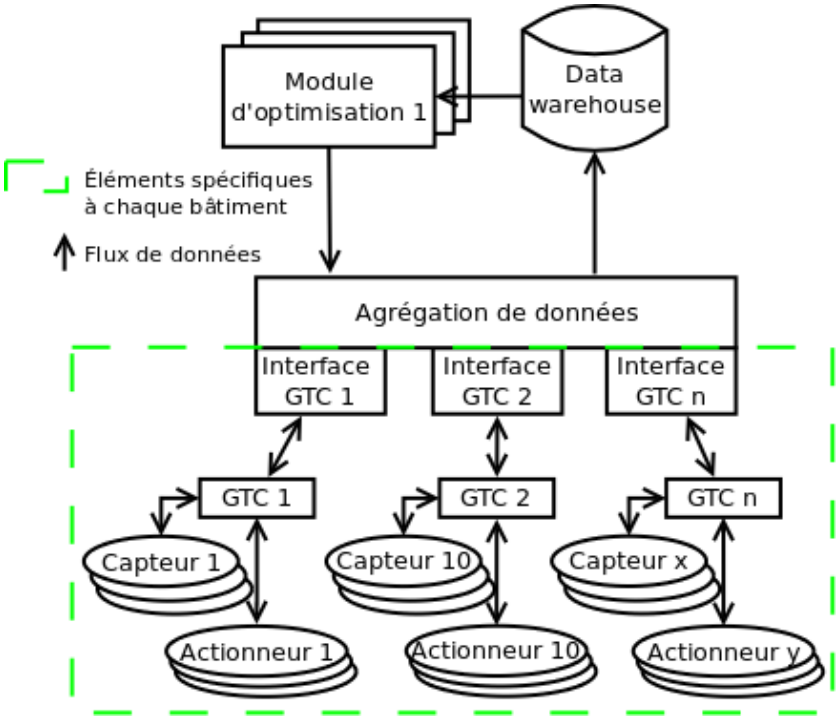

FIGURE 2.3 : Composants de l'architecture logicielle

2.4.1 Usages d'un modèle de bâtiments

Un modèle de bâtiments, aussi appelé maquette numérique, permet de centraliser la description de la structure et des éléments qui le composent [Brad Brech *et al.*, 2011]. Il normalise la description des plans du bâtiment, les équipements disponibles, le format des données sauvegardées, les données nécessaires pour prendre des décisions en temps réel.

Une maquette numérique répond aux besoins suivants :

- Analyse des besoins, par exemple, coûts, qualité,
- Conception, par exemple, conception des réseaux électrique, de chauffage, de ventilation, planification des coûts,
- Construction, par exemple, contrôle des coûts, de la qualité, et du planning,
- Utilisation et maintenance du bâtiment, par exemple, contrôle en temps réel des capteurs et actionneurs, charte de bonnes pratiques, environnement ouvert permettant l'échange d'informations entre les équipes,
- Déclassement et démolition, par exemple, identification des matériaux de valeurs.

Ce modèle joue un rôle de "pivot" entre les différentes sources d'informations disponibles. Cette description centrale sert de référentiel pour centraliser toutes les informations liées à la construction et à la maintenance d'un bâtiment. Il peut être utilisé dès la construction du bâtiment pour choisir une architecture permettant une faible consommation énergétique [Gholipour *et al.*, 2009]. Dans ce cadre, le modèle de bâtiments est utilisé comme référentiel entre les différents acteurs du projet de construction (maître d'œuvre, architecte,

entrepreneur) [Halin et Kubicki, 2005]. Chaque acteur utilisant un logiciel approprié à son métier, lui donnant une “vue” différente sur les données stockées par l'instance du modèle de bâtiments [Halin et Kubicki, 2007; Kubicki *et al.*, 2006]. L'utilisation d'un tel modèle nécessite un effort de standardisation des données et protocoles d'échanges [António Grilo et Ricardo Jardim-Goncalves, 2011]. La spécification Industry Foundation Classes (IFC) définie par l'organisation buildingSMART [buildingSMART, 2013a] vise à simplifier l'interopérabilité entre logiciels de CAO [Jim Steel *et al.*, 2010; buildingSMART, 2013b].

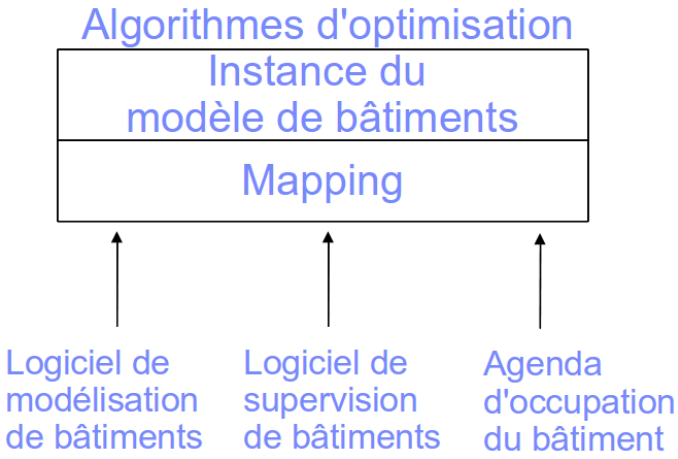

FIGURE 2.4 : Usage d'un modèle de bâtiments

Comme montré dans la figure 2.4, les algorithmes d'optimisation énergétique sont basés sur les données issues d'une instance du modèle de bâtiments. Les données qui sont présentes dans cette instance de modèle sont issues de logiciels différents. Les plans du bâtiment, les équipements présents (par exemple, capteurs, centrales de traitement d'air, actionneurs) sont décrits à l'aide d'un logiciel de conception assistée par ordinateur (CAO). Des logiciels de supervision permettent de fournir en temps réel des données issues des capteurs et actionneurs. D'autres sources d'information peuvent être utilisées pour enrichir la connaissance du bâtiment. Par exemple, il est possible d'utiliser un planning d'occupation des pièces ou un service de prévision météo. L'échange d'informations entre la description du bâtiment et les autres sources d'information est permis par un mapping entre le format de données décrit par le modèle de bâtiments et les formats de données des autres logiciels.

2.4.2 Modèles de bâtiments existants

Il existe de nombreux modèles de bâtiments, chacun cible un usage différent :

- *Industry Foundation Classes* (IFC) [buildingSMART, 2011] permet la conception et la gestion d'un bâtiment. Cela inclut par exemple, l'étude de faisabilité, la construction, et la maintenance [buildingSMART, 2013c].
- *Green Building XML* (gbXML) [gbXML.org, 2013] a pour but de simplifier l'échange des informations de bâtiments entre les logiciels de CAO.

- *Neutral Building Data Model* (NBDM) [NBDM, 2008] est un langage simplifié pour la représentation de bâtiments. Il vise à faciliter l'échange de données entre logiciels de simulation énergétique.
- *Common Data Model* (CDM) [Ling Tai *et al.*, 2008] a comme premier objectif la modélisation de data centers. Il permet notamment la modélisation des salles machines, la configuration matérielle et logicielle des serveurs.
- *City Geography Markup Language* (cityGML) [Gerhard Gröger *et al.*, 2012] est utilisé pour représenter des modèles 3D de villes et de paysages. Il sert à décrire les caractéristiques des terrains, des bâtiments, des plans d'eau, des zones de circulation, de la végétation et du mobilier urbain.
- *Open Building Information Exchange* (oBIX) [OASIS] est un standard de communication entre systèmes de gestion technique centralisée (GTC) de bâtiments. Il se focalise sur l'échange des données de points de mesures (capteurs et actionneurs), les alertes, et l'historique des mesures.

Le tableau 2.1 présente une synthèse des concepts présents dans les différents méta-modèles de bâtiments. Il existe des possibilités de conversion d'un format à l'autre (par exemple, IFC vers gbXML), certains modèles peuvent référencer des données dans un autre (par exemple, gbXML vers IFC), et des possibilités d'extensibilité en spécialisant le modèle (par exemple, les "Application Schemas" utilisés par la norme GML sur laquelle est basé cityGML). Nous devons noter que tous les modèles sont décrits notamment par un schéma XML. Cela permet de simplifier l'échange de données entre les logiciels, mais également d'intégrer la modélisation de bâtiments dans des méthodologies d'ingénierie des modèles [Steel *et al.*, 2011] et ainsi faciliter les conversions entre modèles. Cette intégration permettrait à terme de simplifier des opérations sur un ou plusieurs modèles (par exemple, fusion de différents modèles) et la génération d'outils directement adaptés au bâtiment modélisé.

TABLE 2.1 : Récapitulatif des concepts des différents méta-modèles de bâtiments

Concepts	**IFC**	**gbXML**	**NBDM**	**CityGML**	**CDM**	**oBIX**
Plan bâtiment	C [1]	C	C	C	C	- [2]
HVAC [3]	C	P [4]	-	-	-	-
Capteurs	C	C	-	-	P	P
Actionneurs	C	P	-	-	P	P
Planning de présence	-	P	-	-	-	P
Météo	-	C	-	-	-	P
Environnement	P	P	-	C	-	-
Data centers	-	-	-	-	C	-
Historique	-	-	-	-	-	C
Données temps réel	-	P	-	-	P	C
Représentation 3D	C	C	C	C	P	-
Format XML	C	C	C	C	C	C

1. Support complet
2. Non supporté

Aucun modèle de bâtiments ne permet actuellement de remplir tous les critères. Or, dans RIDER, nous avons besoin d'un modèle qui remplisse tous ces critères car le projet a pour objectif l'optimisation de domaines connectés à ces éléments.

Ces modèles de bâtiments sont des modèles simplifiés de ceux que l'on retrouve dans le domaine de l'ingénierie de la connaissance. Studer *et al.* [1998] présentent un récapitulatif des usages et frameworks utilisés pour modéliser la connaissance relative à un domaine précis, et déduire de nouvelles informations ou prendre des décisions à l'aide d'un moteur d'inférences. Ces approches peuvent être intégrées aux architectures sensibles au contexte présentées dans la section précédente. Les modèles d'ingénierie de la connaissance sont importants pour comprendre les concepts spécifiques à un domaine, leurs interrelations et les tâches associées à la gestion de ces concepts. Ces modèles peuvent être réalisés à l'aide d'ontologies ou du langage UML. Abdullah *et al.* [2005] utilisent un profil UML pour représenter dans un modèle UML les informations relatives la modélisation de la connaissance (par exemple, les concepts d'un domaine, les liens entre concepts, les tâches).

Les outils de modélisation utilisés dans le domaine du bâtiment pourraient bénéficier de ce type d'approche. C'est pourquoi, nous proposons dans la section suivante un nouveau modèle de bâtiments en UML représentant les principaux concepts du domaine. Si nécessaire, ce modèle peut être utilisé avec un moteur d'inférence pour déduire de nouvelles informations ou décrire des consignes de gestion à l'aide de règles.

2.4.3 Un nouveau méta-modèle de bâtiments pensé pour l'optimisation énergétique

Notre nouveau méta-modèle doit répondre aux exigences suivantes :

- permettre la modélisation statique :
  - de bâtiments,
  - des équipements installés (par exemple, les centrales de traitements d'air, chauffages),
  - des systèmes de distribution d'air et d'eau,
  - des capteurs et actionneurs,
  - des data centers,
  - de l'environnement du bâtiment,
  - de méta-données relatives aux éléments d'un bâtiment.
- permettre la modélisation dynamique :
  - de mesures de capteurs, de valeur de consignes qui devraient être mesurées, de prédictions de mesures,
  - des ordres envoyés aux actionneurs
  - des prévisions et relevés météorologiques,
  - des prévisions et relevés de présence dans les pièces,
  - de méta-données relatives aux éléments d'un bâtiment.

3. Heating Ventilation Air-Conditioning
4. Support partiel

Pour réaliser ce nouveau méta-modèle, nous nous sommes basés sur les modèles IFC, gbXML, NBDM, et CDM, ainsi que sur de nombreux entretiens avec des experts du domaine du bâtiment. Nous avons réalisé une quinzaine d'entretiens avec des experts en modélisation et optimisation énergétique, des éditeurs de GTC et des gestionnaires de bâtiments. Nous avons validé notre approche durant des réunion de présentation avec ces mêmes personnes durant lesquelles nous présentions des cas d'utilisation du modèle.

La figure 2.5 présente les concepts de base du méta-modèle de bâtiments. La classe *PhysicalElement* représente tout élément physique, concret, d'un bâtiment (par exemple, un mur, un matériel de chauffage). Un élément physique peut être placé à un emplacement dans un bâtiment. Cet emplacement est représenté par la classe *PhysicalLocation*. Un emplacement représente tout type de site, bâtiment, espace (par exemple, salle de réunion, chambre, bureau, data center). La classe *PhysicalLocation* est associée à la classe *Environment* qui décrit l'environnement extérieur d'un emplacement. Elle permet de décrire, par exemple, le climat, la météo, des éléments extérieurs qui peuvent avoir un impact sur l'efficacité énergétique d'une pièce (des arbres, des constructions avoisinantes, un plan d'eau).

Un système, représenté par la classe *ConcreteSystem* représente une agrégation d'éléments physiques permettant de réaliser des fonctionnalités données. Par exemple, un système peut regrouper les éléments relatifs à un système de distribution d'eau (par exemple, les pompes, tuyaux, capteurs de débits). L'association *GroupElements* décrit les éléments physiques regroupés dans un système.

La classe *PhysicalPackage* étend la classe *PhysicalElement* et représente un contenant, un paquetage physique. Par exemple, il peut s'agir d'une centrale de traitement d'air, d'un matériel de chauffage.

Les services, représentés par la classe *Service* sont des concepts de haut niveau tels qu'un service de climatisation, un service de traitement de l'eau. Un service peut utiliser un ou plusieurs systèmes pour remplir sa tâche. Un service est fourni par un paquetage physique.

La figure 2.6 présente les classes utilisées pour représenter les plans d'un bâtiment du niveau de la pièce, jusqu'au site.

Les classes *Site*, *Building*, *Storey* et *Space* sont reliées par des associations de type agrégation, et héritent toutes de la classe *PhysicalLocation*. La classe *Space* représente un espace, une aire ou un volume borné (par exemple, une pièce). Un espace est associé à un étage ou, dans le cas d'un espace extérieur (par exemple, un parking, une terrasse), à un site. Un espace peut regrouper plusieurs autres espaces connexes, et peut être décomposé en parties.

La figure 2.7 présente les classes relatives à l'instrumentation d'un bâtiment. La classe *InstrumentationElement* décrit tout type d'instrument tels que des capteurs ou des actionneurs. Un instrument peut avoir différents états, décrits par la classe *State*, exprimés à l'aide d'un type de données.

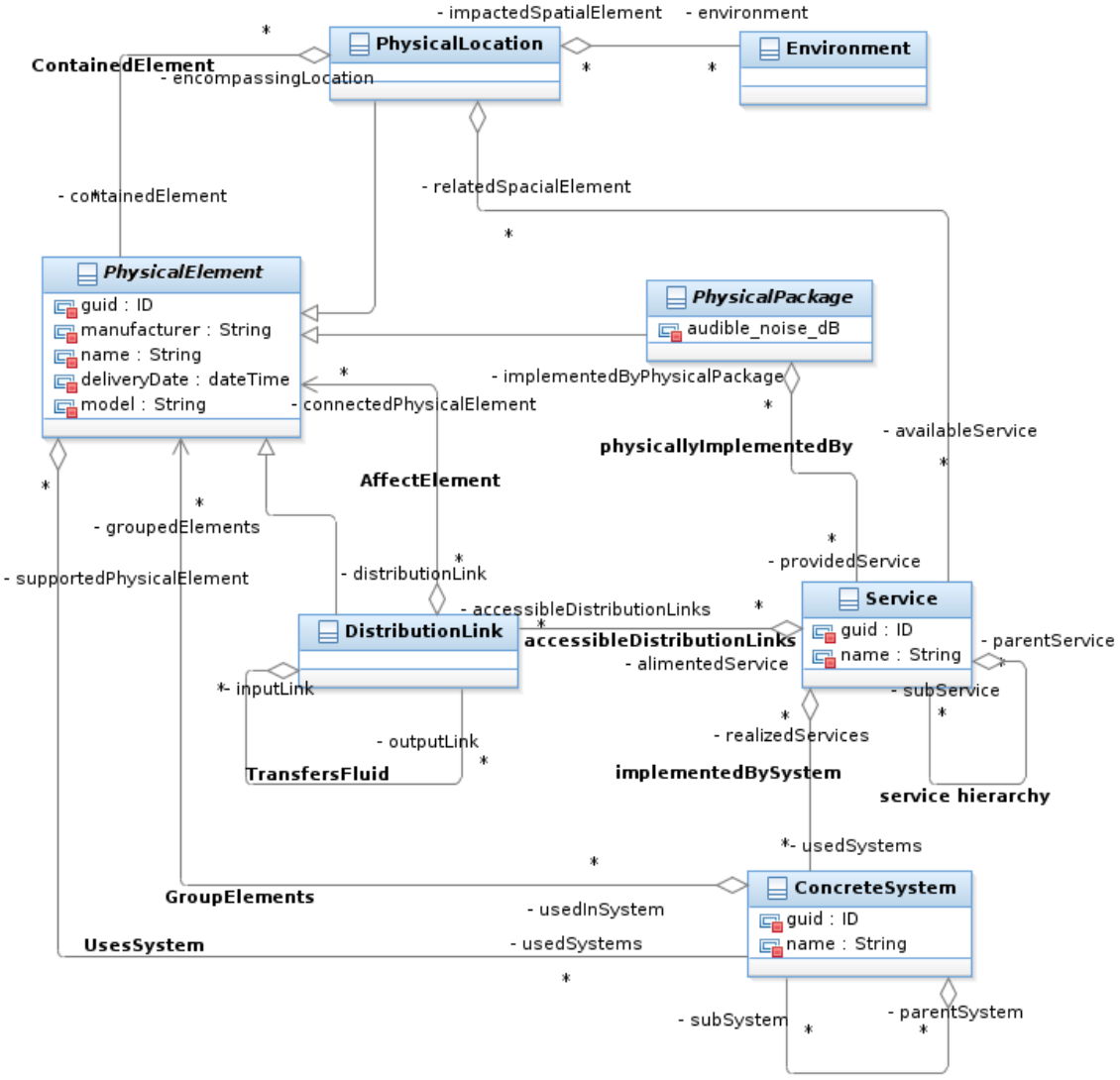

FIGURE 2.5 : Concepts de base du modèle de bâtiments

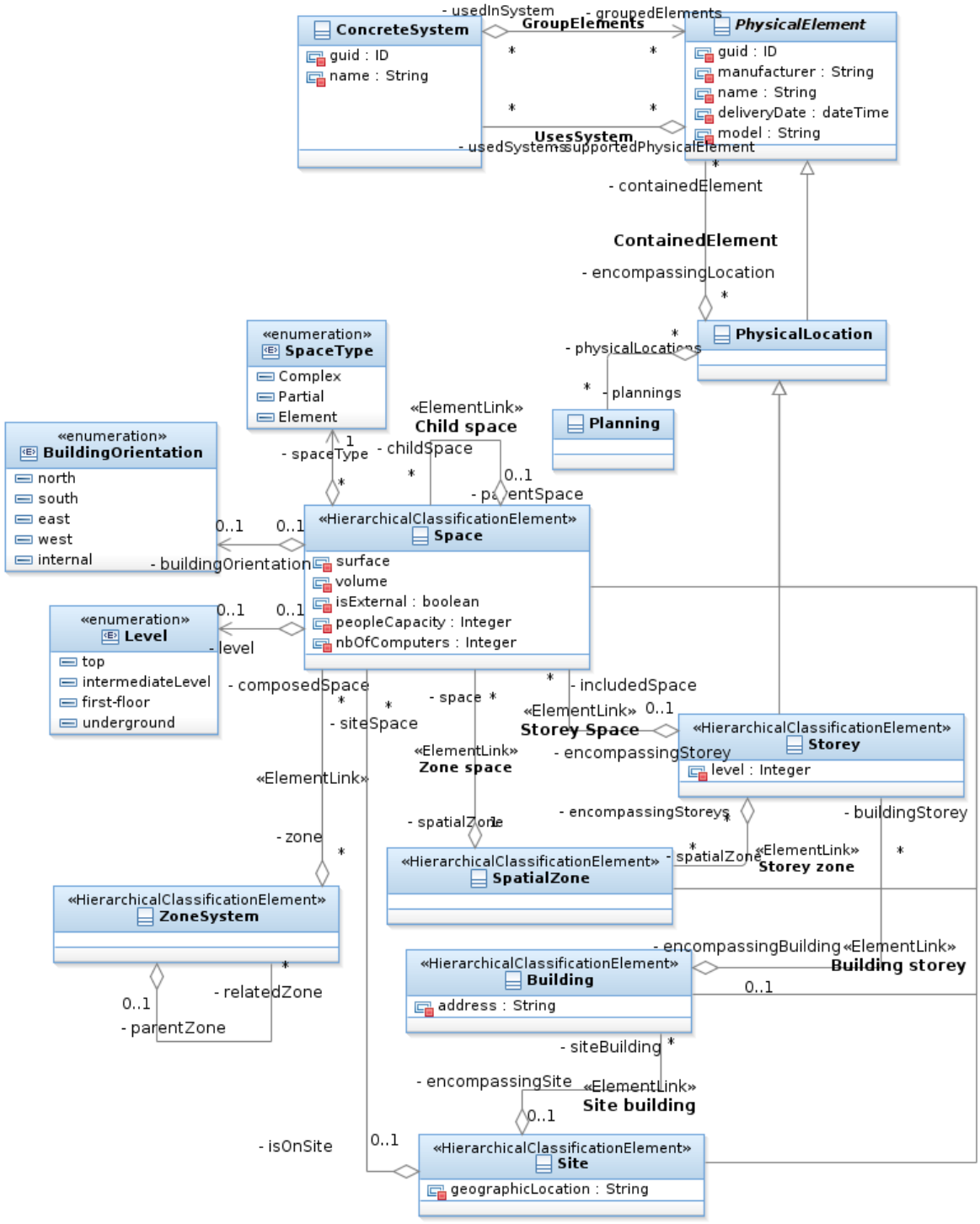

FIGURE 2.6 : Représentation spatiale d'un bâtiment

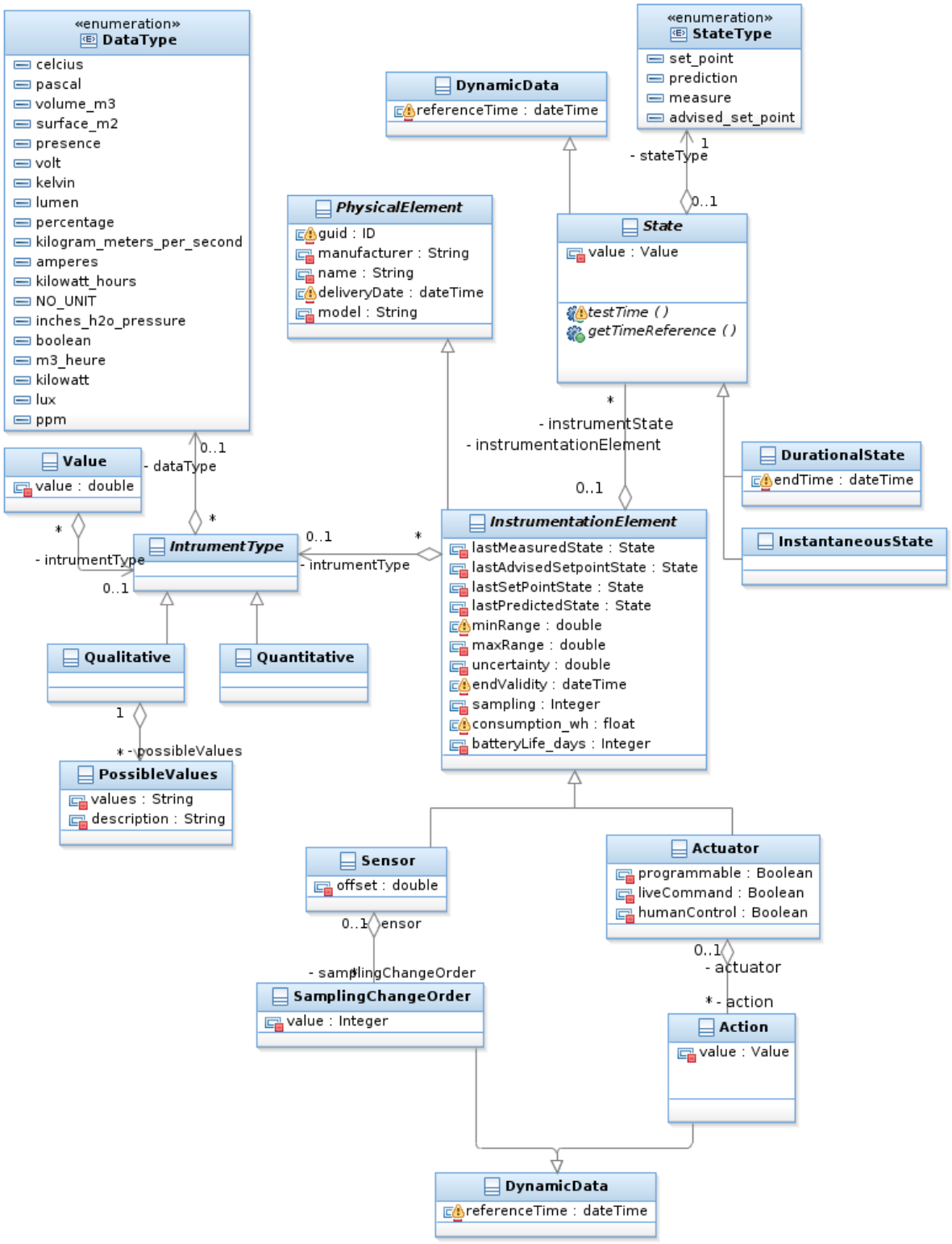

FIGURE 2.7 : Instruments permettant la supervision d'un bâtiment

Si le type de données est qualitatif, la valeur décrite par la classe *Value* représente l'indice de la valeur dans une liste de valeurs possibles décrites par la classe *PossibleValues*.

Le type de l'état d'un instrument est décrit par l'énumération *StateType* :

- *set_point* : une valeur de consigne attendue par l'habitant,
- *prediction* : l'état a été prédit par un calcul ou un algorithme,
- *measure* : l'état est le résultat d'une mesure ou d'une observation,
- *advised_set_point* : l'état représente une valeur de consigne recommandée à l'habitant, qui permettrait de faire des économies d'énergies si elle était appliquée.

L'état peut être instantané (par exemple, une température) ou s'étaler sur une durée (par exemple, une présence dans un bureau).

La figure 2.8 présente les éléments permettant de modéliser les réseaux de distribution d'énergie dans un bâtiment. La classe *DistributionLink* représente un élément capable de transporter un flux d'énergie, un liquide ou un gaz. Cet élément peut affecter un ou plusieurs autres éléments physiques (*PhysicalElement*). Par exemple, un tube transportant de l'eau chaude peut chauffer un espace. Cette relation est représentée par le rôle *connectedPhysicalElement*.

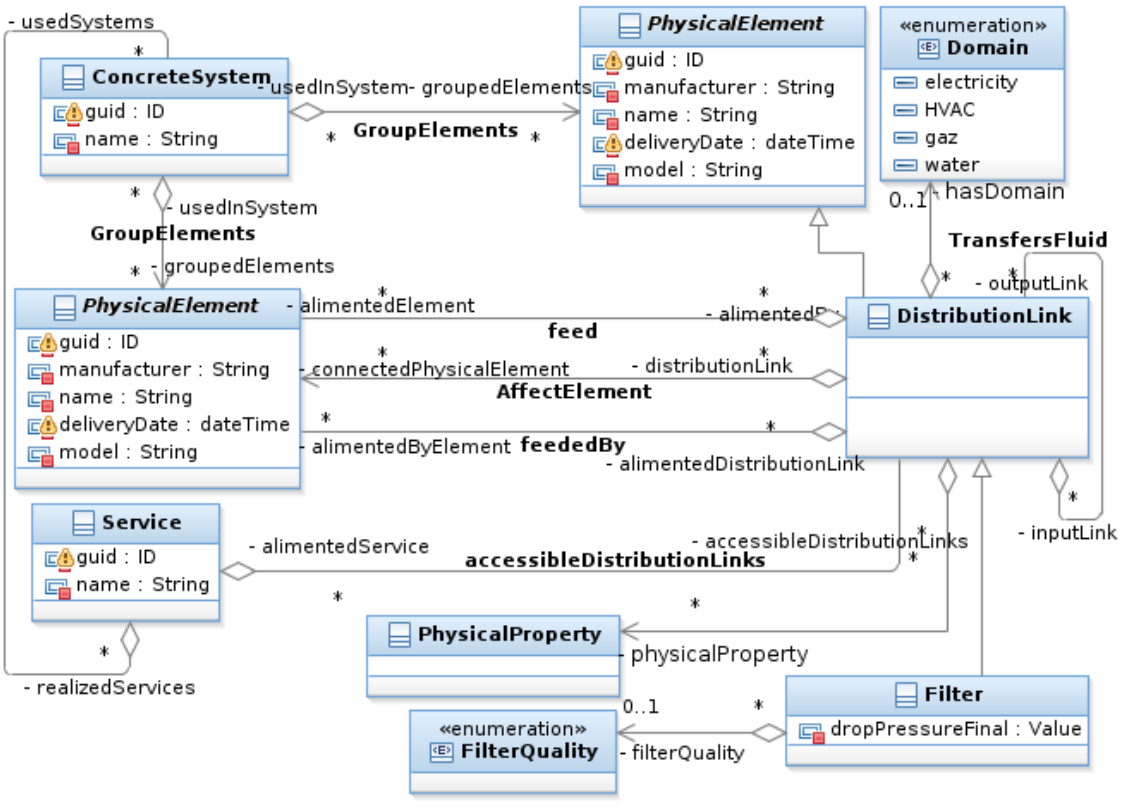

FIGURE 2.8 : Éléments permettant la distribution d'énergie dans un bâtiment

L'association *accessibleDistributionLinks* permet de lier différents services. Par exemple, un service de chauffage peut être lié à un service de distribution électrique. Un réseau de

distribution d'énergie ne peut alimenter que des services qui utilisent l'un des systèmes auxquels appartient le réseau.

Le rôle *alimentedElement* décrit quels éléments utilisent le flux apporté par un *DistributionLink*. Par exemple, une pièce utilise l'air fourni par un système de ventilation. Le rôle *alimentedByElement* décrit quels éléments alimentent le flux transmis par un *DistributionLink*. Ou encore, une centrale de traitement d'air alimente un système de ventilation.

La figure 2.9 présente les classes permettant d'ajouter des méta-données à la représentation du bâtiment. Cela permet aux différents composants d'optimisation d'enrichir les informations du bâtiment avec les données qu'ils calculent.

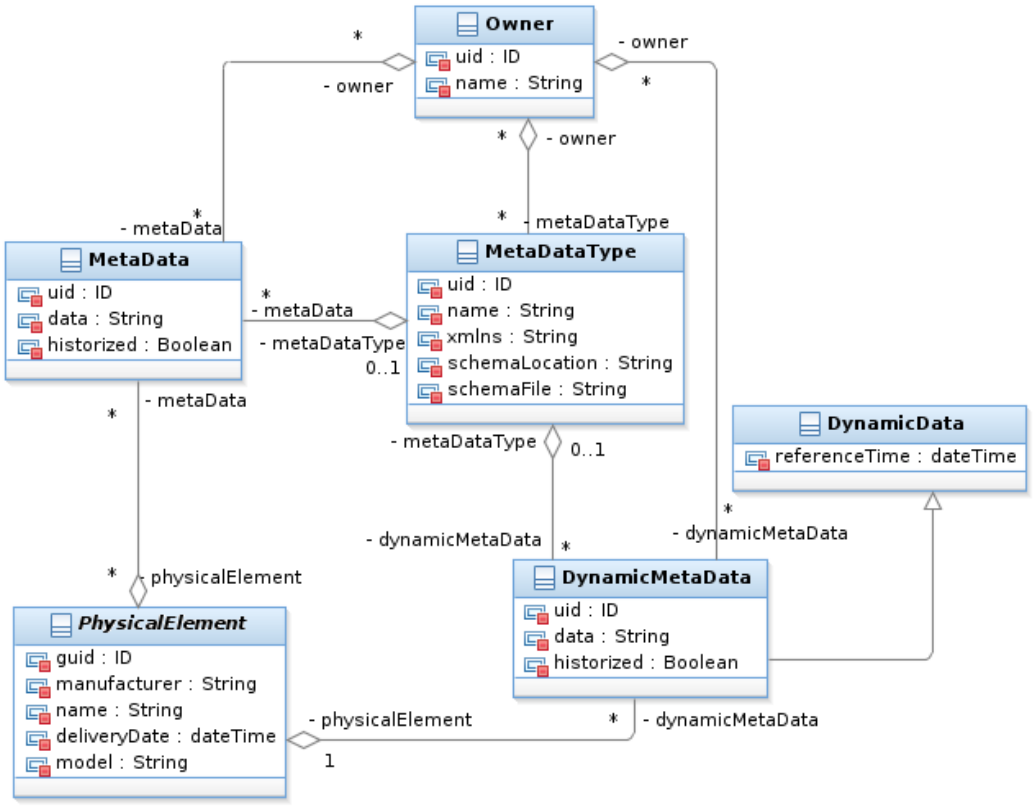

FIGURE 2.9 : Méta-données associées aux éléments qui décrivent un bâtiment

Une méta-donnée est une instance de la classe *MetaData*. L'information est représentée dans le format XML. Ainsi, le type de la méta-donnée est fourni par la classe *MetaDataType* dont une instance référence le schéma XML d'un type de méta-données. La classe *Owner* permet l'identification du propriétaire des méta-données. Nous faisons la distinction entre méta-données dynamiques (classe *DynamicMetaData*) et statiques (classe *MetaData*). Cela permet de traiter différemment les données dynamiques dont l'historique ne doit pas nécessairement être conservé.

Comme précisé par Reichle *et al.* [2008], les méta données sont utilisées pour enrichir les informations concernant le modèle de contexte. Ils associent les méta-données à des concepts et des valeurs du contexte, et les décrivent à l'aide d'un schéma de représentation spécifié dans le modèle contenant les méta-données. Ils considèrent qu'il est plus pratique d'intégrer les méta-données dans le modèle de contexte pour les traiter et les actualiser en même temps que les données qu'elles enrichissent.

Nous n'avons pas décrit ici le détail des spécialisations de classes qui représentent les différents types de capteurs, d'actionneurs, et d'équipements pour ne pas alourdir les diagrammes.

Les concepts que nous avions identifiés dans la table 2.1 sont implémentés par notre nouveau modèle de bâtiments :

- Le plan du bâtiment et sa représentation 3D sont modélisés par les classes que nous avons décrites dans la figure 2.6.
- Les systèmes de chauffage, ventilation et climatisation (HVAC) sont également supportés. Les réseaux de distribution d'air, d'eau et d'électricité sont décrits par les classes présentées dans la figure 2.8. Les systèmes HVAC, tels que les centrales de traitement de l'air sont représentés par des spécialisations de la classe *PhysicalPackage*.
- Les capteurs et actionneurs sont modélisés par les classes présentées dans la figure 2.7. Les différents types de capteurs (par exemple, température, humidité) sont des spécialisations de la classe *Sensor*. Les différents types d'actionneurs (par exemple, vannes, ventilateurs) sont des spécialisations de la classe *Actuator*.
- Le planning d'occupation des pièces est décrit par des prédictions de mesures d'un capteur de présence.
- La météo et l'environnement sont décrits par la classe *Environment*.
- Les serveurs et systèmes de refroidissement d'un data center sont décrits par des classes qui spécialisent *PhysicalElement* et *PhysicalPackage*.
- L'historique et la gestion des données en temps réel sont pris en charge par les classes décrites dans la figure 2.7.
- L'export du modèle en XML est simplifié car il est décrit à l'aide du langage UML.

2.5 Conclusion

Dans ce chapitre, nous avons décrit brièvement comment un logiciel d'optimisation est mis en œuvre pour améliorer l'efficacité énergétique d'un bâtiment ou d'un groupe de bâtiments. Nous avons passé en revue les méthodes d'optimisation les plus courantes, ainsi que l'infrastructure matérielle et les architectures logicielles permettant de les mettre en œuvre.

Nous avons décrit l'architecture logicielle sur laquelle la suite de nos travaux s'appuie. Pour permettre sa mise en œuvre (qui sort du cadre de cette étude) nous avons proposé un nouveau modèle de bâtiments permettant de palier les manques identifiés dans les modèles existants.

Cependant, la mise en œuvre d'un logiciel d'optimisation énergétique dans un bâtiment

nécessite l'identification des types d'optimisation qui pourront être appliqués. L'instrumentation de certaines zones et les attentes des utilisateurs peuvent contraindre le choix du type d'optimisation. Cela peut conduire à utiliser des méthodes d'optimisation différentes dans un même bâtiment, et de façon plus générale, impose une adaptation du logiciel pour chaque installation dans un bâtiment.

CHAPITRE

3 Introduction aux lignes de produits logiciels et modèles de caractéristiques

Sommaire

3.1 Les lignes de produits logiciels et leur cycle de vie **28**
3.2 Les différents types de variabilité **29**
3.3 Modèles de caractéristiques **32**
3.4 Conclusion **37**

Notre contribution se plaçant dans le contexte des lignes de produit, nous présentons en premier lieu ce contexte de développement de façon générale, puis nous introduisons plus précisément le concept de “modèle de caractéristiques”

Nous présentons tout d’abord dans la section 3.1 les lignes de produits logiciels de façon générale et leur cycle de vie. Nous décrivons dans la section 3.2 les différents types de variations qui peuvent être rencontrés parmi un ensemble de produits d’une même famille. Ensuite, nous définissons dans la section 3.3 ce qu’est un modèle de caractéristiques, les différents usages possibles avec les méta-modèles existants, et les différentes approches permettant d’adapter un modèle de caractéristiques selon l’environnement d’exécution d’un futur produit. La section 3.4 présente les problématiques que nous avons identifiées et que nous allons aborder dans la suite de ce mémoire.

3.1 Les lignes de produits logiciels et leur cycle de vie

Une ligne de produits logiciels est un ensemble de logiciels qui partagent des propriétés communes satisfaisant les besoins d'un domaine métier particulier, et qui sont développés à partir d'un ensemble commun de composants logiciels [Linda M. Northrop et Paul C. Clements, 2007; ISO/IEC/IEEE, 2010; Clements *et al.*, 2005].

La réalisation d'une ligne de produits se base sur deux processus distincts : l'*ingénierie du domaine* (*domain engineering*), et l'*ingénierie de l'application* (*application engineering*) [Böckle *et al.*, 2005; Klaus Pohl *et al.*, 2005].

Le processus d'*ingénierie du domaine*. Ce processus vise à créer et organiser l'ensemble des artefacts (par exemple, documents, code, composants logiciels, librairies, ...), aussi appelés *assets*, nécessaires pour produire des logiciels dans un domaine clairement identifié (par exemple, firmware d'imprimante, logiciel de gestion d'entreprise, système d'exploitation de téléphone portable, ...). Ces artefacts sont organisés de façon à pouvoir déterminer lesquels sont indispensables, interchangeables ou dépendants d'autres, dans le cadre d'une même architecture logicielle.

Le processus d'*ingénierie de l'application*. Ce processus consiste à réutiliser les artefacts existants pour produire un nouveau produit répondant à des besoins spécifiques. L'étape du choix des artefacts à réutiliser s'appelle aussi la *configuration d'un produit*. Si certains besoins ne peuvent pas être satisfaits par les artefacts existants de la ligne de produits, de nouveaux artefacts sont développés et sont ensuite intégrés dans la ligne de produits pour une réutilisation future.

Le choix des différents artefacts constituant un nouveau produit est soumis à des contraintes de variabilité. La variabilité d'une ligne de produits est généralement exprimée à l'aide d'un modèle de caractéristiques [Istoan *et al.*, 2011]. Une caractéristique décrit tout aspect, ou fonctionnalité visible pour les parties prenantes d'un système [Kang *et al.*, 1990; Bontemps *et al.*, 2004; Schobbens *et al.*, 2007]. Un modèle de caractéristiques a pour but de représenter l'ensemble des caractéristiques d'une ligne de produits à l'aide de :

- diagrammes de caractéristiques, représentant hiérarchiquement les caractéristiques de la ligne de produits,
- contraintes déterminant la variabilité de la ligne de produits.

Les contraintes peuvent notamment être liées à la compatibilité des différents composants logiciels (par exemple, dans un système d'exploitation de téléphone, une caractéristique *appareil photo* nécessite une caractéristique *galerie photos*). Un point de variabilité rend possible un choix entre plusieurs caractéristiques. Cela permet de décrire les caractéristiques communes à tout produit, et les caractéristiques décrivant des variantes possibles entre les produits issus de la ligne de produits. La variabilité au sein d'un modèle de caractéristiques sera décrite plus loin dans ce chapitre dans la section 3.3.

Le *Common Variability Language* (CVL) [SINTEF, 2012; Fleurey *et al.*, 2009] a pour objec-

tif de proposer une approche par ingénierie des modèles pour générer des produits décrits par un méta-modèle et dont les contraintes de variabilité sont décrites dans un modèle de caractéristiques. Il permet de représenter les liens entre le modèle de caractéristiques et un modèle représentant les artefacts et leurs interrelations (par exemple, un schéma d'architecture logicielle). CVL est encore en cours de standardisation par l'OMG mais permettra à terme de gérer une ligne de produits et d'automatiser la configuration de nouveaux produits.

Nous ne traitons pas dans le cadre de cette thèse l'intégration de ces deux processus dans les cycles de développement d'un logiciel. Cet aspect est décrit notamment par Linda M. Northrop et Paul C. Clements [2007], Clements *et al.* [2005] et Klaus Pohl *et al.* [2005]. Nous nous concentrons, dans la suite de cette introduction, sur la description de la variabilité au moyen d'un modèle de caractéristiques dans le processus de domain engineering, et à sa configuration dans le processus du product engineering.

3.2 Les différents types de variabilité

Cette section identifie les types de variations entre des produits issus d'une même ligne de produits.

Les variations possibles sont identifiées par des *points de variabilité*. Un point de variabilité donne le choix entre plusieurs *variantes*. Chaque variante est implémentée par un ou plusieurs artefacts différents [Klaus Pohl *et al.*, 2005].

Les artefacts composant la ligne de produits peuvent évoluer de différentes manières durant le développement des produits. Klaus Pohl *et al.* [2005] définissent la variabilité dans le temps et dans l'espace. La variabilité dans le temps est due à la création de nouvelles versions d'un artefact au cours du temps. Cela permet de décrire des artefacts rendus obsolètes par l'arrivée de nouvelles technologies. Par exemple, la norme de formatage de données HTML a eu différentes versions successives, chacune remplaçant la précédente : par exemple, HTML 4.0, XHTML, HTML5.

La variabilité dans l'espace décrit les variantes d'un point de variabilité décrivant les variations d'un même artefact. Par exemple, un point de variabilité *Protocole de transfert de fichiers* possède les variantes *Transfert via FTP* et *Transfert via HTTP*.

D'après Groher et Voelter [2007] il y a deux façons de caractériser la variabilité lors de la réutilisation de modèles, de code source, ou de composants logiciels. Il s'agit de la variabilité *positive* et de la variabilité *négative*. La variabilité positive consiste à ajouter des éléments optionnels à un noyau donné. La variabilité négative consiste à retirer des éléments optionnels à une structure donnée.

Bachmann et Bass [2001] identifient six catégories dans un logiciel sujettes à la variabilité.

La *variabilité de fonctionnalités* concerne des fonctionnalités qui peuvent être présentes dans certains produits, mais pas d'autres. Par exemple, dans le cadre d'un logiciel d'optimisation énergétique de bâtiments, une fonctionnalité *Calcul de l'indicateur d'efficacité énergétique du data center* est présente dans les logiciels utilisés dans des bâtiments équipés d'un data center.

La *variabilité de données* concerne le modèle de données utilisé dans une application. Par exemple, il existe de nombreux standards de représentation de bâtiments (par exemple, Industry Foundation Classes, Green Building XML, ...) et de protocoles de communication avec des capteurs et actionneurs (par exemple, oBIX, Modbus). Une ligne de produits logiciels pour l'optimisation énergétique de bâtiments doit permettre l'acquisition d'informations sur l'architecture d'un bâtiment décrites à l'aide de formats différents, et la communication avec des logiciels de supervision utilisant des protocoles d'échange de données variés.

La *variabilité dans les processus de contrôle* concerne, par exemple, la gestion des alertes dans un logiciel d'optimisation énergétique. Une alerte concernant un capteur déficient peut enclencher un processus au niveau du logiciel de supervision. Ce dernier ne transmet donc plus de données au logiciel d'optimisation. Ou bien, l'alerte peut être traitée au niveau du logiciel d'optimisation, qui invalide alors toutes les mesures effectuées par ce capteur. Le choix est déterminé par la capacité du logiciel de supervision à inhiber l'acquisition de données issue de capteurs défectueux.

La *variabilité en technologies* décrit les différents composants logiciels nécessaires au bon fonctionnement d'un produit. Par exemple, des capteurs intelligents (*i.e.*, capables de stocker leurs mesures, embarquant une capacité de traitement de leurs données, et de communiquer de façon bidirectionnelle avec l'extérieur) peuvent se connecter directement à un logiciel d'optimisation énergétique. Des capteurs conventionnels nécessitent l'utilisation d'un logiciel de supervision faisant l'interface entre les capteurs et le logiciel d'optimisation.

La *variabilité en objectifs de qualité* concerne les différents objectifs de qualité à atteindre. Par exemple, la fréquence de rafraîchissement des tableaux de bord d'un système de data mining, est dépendante de la puissance de calcul du serveur hébergeant l'entrepôt de données.

La *variabilité de l'environnement* décrit comment un produit peut interagir avec son environnement. Par exemple, la présence de différents types de capteurs et de superviseurs nécessite des interfaces de communications adaptées dans le logiciel d'optimisation énergétique.

Günter Halmans et Klaus Pohl [2003] ont étudié comment décrire la variabilité d'une ligne de produits afin d'en simplifier la compréhension. Un système de vues [Boton *et al.*, 2010] sur un modèle de variabilité peut permettre d'adapter la présentation d'un modèle à un rôle particulier. Ils identifient trois rôles nécessaires à la création d'une ligne de produits :

- *Le client* décrit les fonctionnalités dont il a besoin en regard des fonctionnalités dis-

ponibles dans la ligne de produits et détermine si les fonctionnalités présentes dans la ligne de produit correspondent, au moins partiellement, à ses besoins.
- *L'analyste des besoins métiers* documente les besoins du client. Cette documentation permet d'identifier les besoins satisfaits, partiellement satisfaits, ou non satisfaits par les artefacts existants de la ligne de produits. Ainsi, il peut estimer les coûts de développement du produit du client, et si nécessaire, proposer des compromis permettant de réduire les coûts de développement. Cette documentation est destinée à l'ingénieur en charge de la ligne de produits.
- *L'ingénieur en charge de la ligne de produits* assure la faisabilité de nouveaux produits réutilisant des assets existants. Son rôle consiste à s'assurer que les dépendances entre assets sont cohérentes par rapport aux contraintes de fonctionnement de chacune. Il gère également les mécanismes permettant d'implémenter dans un nouveau produit les variantes choisies pour chaque point de variabilité.

Ainsi, le rôle de l'analyste consiste à trouver des compromis entre les besoins du client et les possibilités des assets et des points de variabilité existants dans la ligne de produits. Chacun de ces trois rôles a besoin d'avoir une vision différente de la ligne de produits. Günter Halmans et Klaus Pohl [2003] ont identifié des catégories de variabilité adaptées pour chacun des points de vues énoncés précédemment.

- La *variabilité essentielle* décrit les points de variabilités pertinents pour le client. Il s'agit des variantes dont le choix a un impact directement visible pour le métier du client et des utilisateurs du produit final. Cette catégorie est subdivisée en cinq sous-catégories :
    1. *Les fonctionnalités* décrivent par exemple, dans le cadre d'un logiciel d'optimisation énergétique, la possibilité d'accéder à distance aux tableaux de bord du système, la possibilité de planifier l'occupation des salles du bâtiment.
    2. La variabilité dans *l'environnement du système* concerne par exemple le nombre d'utilisateurs du système, le nombre et le type d'instruments présents dans le bâtiment dont l'énergie est optimisée par le système.
    3. Les variations *d'intégration dans les processus métier* nécessitent la différentiation entre les variations dans les rôles des utilisateurs, et les variations possibles dans les processus métier.
        - Les variations entre les rôles des utilisateurs selon leurs responsabilités concernent par exemple (dans le cadre d'un logiciel d'optimisation énergétique) les droits d'accès aux fonctionnalités d'édition de consignes de température des différentes pièces d'un bâtiment, et aux consignes de réutilisation de l'air des centrales de traitement d'air (CTA).
        - Les variations de processus métier concernent par exemple la procédure à suivre lorsqu'un équipement défaillant est détecté. Un logiciel de supervision équipant un bâtiment avec plusieurs centaines de bureaux pourra envoyer un message à une équipe de maintenance avec toutes les informations liées à l'équipement, voire anticiper les pannes compte tenu d'un planning de maintenance. Le logiciel de supervision d'une maison individuelle se contentera d'un message par mail ou SMS pour informer les occupants de la défaillance d'un équipement.

4. Les variations *qualitatives* concernent par exemple, la sécurité, le niveau de disponibilité, ou le passage à l'échelle du produit.
5. Les variations *d'information, de données* concernent la façon dont le futur produit peut communiquer avec son contexte d'exécution.
   - *La fréquence d'envoi de données.* Par exemple, la fréquence d'envoi de mesures de capteurs influence la qualité des prédictions des besoins énergétiques d'un bâtiment.
   - *La richesse des informations sur l'environnement du produit.* Par exemple, les informations concernant l'âge, les matériaux utilisés, ou les plans d'un bâtiments permettent de réaliser des simulations énergétiques plus précises.
   - *Le méta-modèle des données.* Le format des données peut varier en fonction des logiciels avec lesquels un nouveau produit doit interagir. Les différents superviseurs de bâtiments du marché peuvent utiliser différents méta-modèles pour représenter les informations d'un bâtiment (par exemple, IFC, gbXML, NBDM), et pour stocker et communiquer les mesures des instruments (par exemple, oBIX, accès à une base relationnelle).

- La *variabilité technique* concerne les choix liés à l'implémentation ou à la mise en œuvre d'un nouveau produit. Ces choix ne peuvent pas être effectués directement par le client car ils nécessitent une expertise technique. Cette catégorie est subdivisée en trois sous-catégories :
  - *L'infrastructure informatique* concerne les choix relatifs aux systèmes matériels utilisés. Par exemple, l'utilisation d'un ordinateur central (mainframe) *IBM System z* favorise l'utilisation de certains logiciels pour bénéficier de ses avantages. Ce choix technique peut être favorisé en fonction de critères qualitatifs définis par le client. De plus, des choix matériels (par exemple, cloud, système embarqué) différents sont faits selon la charge et le volume de données attendus.
  - *L'implémentation* des points de variabilité et le mécanisme de mise en œuvre de variantes peuvent être faits de différentes façons. Linda M. Northrop et Paul C. Clements [2007] résument les différents types de mécanismes permettant d'implémenter la variation. Cette étude sort du cadre de cette thèse.
  - Le *binding time* décrit quand un point de variabilité est pris en compte lors de la génération d'un produit. Cela peut-être fait par exemple, lors de la compilation du code source du produit, ou à l'exécution. Le binding time a un impact sur la flexibilité de la ligne de produits. Un point de variabilité dont la variante peut être choisie à l'exécution, peut permettre de reconfigurer un produit à l'exécution contrairement à un point de variabilité dont la variante est choisie à la compilation.

La section suivante présente comment ces différents types de points de variabilité peuvent être représentés.

3.3 Modèles de caractéristiques

Les fonctionnalités des composants logiciels constituant une ligne de produits logiciels (LdP) sont communément représentées avec un *modèle de caractéristiques* [Kang *et al.*,

2002] dont la sémantique est définie par un méta-modèle [Fey *et al.*, 2002; Asikainen *et al.*, 2006] ou une grammaire [Batory, 2005; Schobbens *et al.*, 2007].

Cette section décrit de façon générique ce qu'est un modèle de caractéristiques et ses différents usages. La section 4.2 du chapitre 4 présentera une comparaison détaillée des concepts des méta-modèles de caractéristiques existants. Nous proposerons dans la section 5.3 du chapitre 5 une définition des principaux concepts des lignes de produits.

3.3.1 Définition

Le modèle de caractéristiques décrit l'ensemble des fonctionnalités implémentées par des composants logiciels réutilisables et les contraintes permettant de les assembler et les réutiliser pour créer de nouveaux produits logiciels. Un nouveau produit logiciel est donc construit en choisissant un ensemble de caractéristiques satisfaisant les contraintes du modèle. L'ensemble des caractéristiques sélectionnées est appelé une *configuration de produit*.

Un modèle de caractéristiques a trois usages principaux :

1. Il permet la description des caractéristiques communes et spécifiques des logiciels d'un même domaine [Kang *et al.*, 1990]. Ainsi, il peut être utilisé comme moyen de communication entre le client, l'analyste et le développeur.
2. Le modèle de caractéristiques est analysé algorithmiquement afin de détecter des problèmes dans sa conception [Thomas von der Maßen et Horst Lichter, 2004; Benavides *et al.*, 2010], par exemple, déterminer le nombre de configurations possibles, si certaines caractéristiques ne peuvent pas être sélectionnées.
3. Le modèle permet de valider qu'un ensemble de caractéristiques sélectionnées permettra la création d'un produit fonctionnel.

La sémantique et la présentation des modèles de caractéristiques varient selon les auteurs et ne permettent pas de prendre en compte tous les différents types de variabilité vus précédemment. De même, le mode de représentation, graphique ou non, permettant de d'échanger des informations sur la variabilité entre les différents acteurs du projet de développement logiciel est important. Cet aspect a un impact sur la facilité de communication entre les différents acteurs d'un projet de développement logiciel [Günter Halmans et Klaus Pohl, 2003].

Czarnecki *et al.* [2005] associent une multiplicité à chaque caractéristique pour spécifier le nombre de fois qu'elle peut être dupliquée dans un produit. Par exemple, le modèle de caractéristiques d'un tableau de bord contenant la caractéristique *indicateur* possède la multiplicité *(1, 5)*. Ainsi, un tableau de bord doit contenir au minimum un indicateur, et au maximum 5. La borne maximale est donnée par une contrainte technique de l'interface graphique du tableau de bord.

3.3.2 Les usages des modèles de caractéristiques

Le tableau 3.1 récapitule les différents critères permettant de caractériser les usages des méta-modèles de caractéristiques existants.

TABLE 3.1 : Les différents types de variabilité (exception faite de la variabilité entre caractéristiques) et modèles de caractéristiques

Méta-modèle	**Variabilité de l'environnement**	**Variabilité de données**	**Identification des rôles**	**Groupement des caractéristiques**	**Choix des caractéristiques pertinentes à chaque rôle**
FODA [Kang *et al.*, 1990]	-	-	-	-	-
FORM [Kang *et al.*, 1998]	-	-	-	oui – *layers*	-
Halmans et al. [2003]	-	-	oui – *Acteurs*	oui – *UML packages*	oui – *associations*
Fey *et al.* [2002]	-	-	-	oui – *feature sets*	-
Czarnecki *et al.* [2005]	partiellement	partiellement	-	-	oui - spécialisation du modèle de caractéristiques
Acher *et al.* [2009, 2011a]	partiellement	partiellement	-	-	-

Variabilité de l'environnement. Un produit peut être configuré en fonction de son environnement, ou contexte, d'exécution. L'environnement d'exécution peut être modélisé sous la forme d'un modèle de caractéristiques [Acher *et al.*, 2009; Jaroucheh *et al.*, 2010b], ou d'une ontologie [Jaroucheh *et al.*, 2010a]. La configuration d'un modèle de caractéristiques peut être contrainte par son contexte [Fernandes *et al.*, 2008, 2011]. Cela est particulièrement utile dans le cas des lignes de produits dynamiques qui doivent être reconfigurées automatiquement (cela sort du cadre de cette étude).

Variabilité de données. La disponibilité de données peut varier dans l'environnement. Par exemple, dans le cadre d'un logiciel d'optimisation énergétique, la présence de certains capteurs détermine la disponibilité de données durant l'exécution du produit dont l'absence peut compromettre certaines fonctionnalités.

Identification des rôles. Différentes parties prenantes peuvent être impliquées dans un projet de développement logiciel. Chacune a un rôle et des compétences différentes. Ainsi, le choix lié à certains points de variabilité doit être fait par une partie prenante ayant un rôle précis. Par exemple, un client dont l'expertise porte uniquement sur son domaine métier peut choisir des caractéristiques ayant un impact direct sur son métier, mais ne peut pas choisir des caractéristiques liées à l'implémentation technique d'un logiciel.

Groupement des caractéristiques. Les groupes de caractéristiques permettent de structurer un modèle de caractéristiques. Ces groupes peuvent être définis selon le niveau d'abstraction des caractéristiques ou servir à grouper des caractéristiques selon la nature des fonctionnalités qu'elles représentent.

Choix des caractéristiques pertinentes à chaque rôle.
Ce critère décrit comment les choix effectués par les différentes parties prenantes de la conception d'un nouveau produit peuvent influer les uns sur les autres. Par exemple, un logiciel peut être créé à partir d'une agrégation de composants logiciels configurés par des fournisseurs différents.

La méthode FODA [Kang *et al.*, 1990] est la méthode originale utilisant et définissant les modèles de caractéristiques. Les possibilités offertes par cette méthode ne répondent pas aux critères énoncés précédemment.

La méthode FORM [Kang *et al.*, 1998] est une amélioration de la méthode FODA. Elle permet de grouper des caractéristiques dans des *couches* ou *layers* pour séparer les différents niveaux d'abstraction représentés par les caractéristiques.

Günter Halmans et Klaus Pohl [2003] proposent de décrire la variabilité d'une ligne de produits à l'aide de diagrammes de cas d'utilisation. Cette approche permet d'identifier les rôles des parties prenantes liées à la ligne de produits à l'aide de la classe UML *Acteurs*. Ainsi, les caractéristiques, décrites ici par des cas d'utilisation, peuvent être groupées dans des paquetages UML. Les caractéristiques pertinentes à chaque rôle sont identifiées par les associations entre acteurs et cas d'utilisation.

Fey *et al.* [2002] groupent les caractéristiques à l'aide de *feature-sets*. Contrairement aux groupements précédents qui imposent une structure hiérarchique (paquetages, couches), les *feature-sets* permettent des groupements arbitraires de caractéristiques.

Czarnecki *et al.* [2005] ne supportent que partiellement la variabilité de l'environnement. En effet, l'adaptation à l'environnement ne peut être obtenue que manuellement en éditant les multiplicités des caractéristiques. Les auteurs permettent aux différentes parties prenantes de configurer séparément un modèle de caractéristiques grâce à la spécialisation. Par exemple, des entreprises peuvent livrer à leurs clients des composants logiciels configurés différemment (par exemple, avec des interfaces différentes). Les clients peuvent réutiliser ces composants logiciels dans leur propre ligne de produits si nécessaire en spécialisant à nouveau les composants fournis.

Acher *et al.* [2009, 2011a] proposent de représenter l'environnement des produits sous la forme d'un modèle de caractéristiques dans lequel chaque choix de caractéristique est déterminé par un événement extérieur. Les caractéristiques de l'environnement sont liées par des contraintes aux caractéristiques de la configuration du produit. Ainsi, chaque changement dans l'environnement entraîne une reconfiguration du produit. Cette approche ne permet pas d'adapter la construction du modèle de caractéristiques du produit en fonction du contexte. Par exemple, dans le cadre d'un logiciel d'optimisation énergétique de bâtiment, cette approche ne permettrait pas d'adapter le modèle de caractéristiques aux spécificités du bâtiment car certains caractéristiques du logiciel doivent être dupliquées pour chaque occurrence d'équipement ou de lieux.

Nous ne comparons pas ici CVL aux autres approches car CVL n'utilise pas de modèle de caractéristiques, mais un "modèle décisionnel" [Czarnecki *et al.*, 2012]. De même, nous

n'étudions pas ici l'ensemble des approches liées à la reconfiguration dynamique de modèles de caractéristiques [Morin *et al.*, 2010] car cela sort du cadre cette étude. Cependant, nous prenons en compte de quelles façons ces approches lient le modèle de caractéristiques à son contexte d'exécution, et comme nous le verrons dans la section suivante, comment est adapté le modèle de caractéristiques en fonction de son contexte [Fernandes *et al.*, 2008].

Czarnecki *et al.* [2004b] proposent de configurer un modèle de caractéristiques en plusieurs étapes. Chaque étape vise à réduire le nombre de choix possibles en choisissant ou en écartant des caractéristiques, ou en réduisant leurs multiplicités [Czarnecki *et al.*, 2004a]. Ainsi, leur approche permet de supporter partiellement la variabilité de l'environnement et des données car les multiplicités des caractéristiques peuvent être adaptées à chaque spécialisation du modèle de caractéristiques afin d'adapter le produit au nombre d'occurrences d'éléments du contexte.

3.3.3 Adaptation de modèles de caractéristiques

Dans le cas de lignes de produits comportant un nombre important de caractéristiques, ou un nombre important d'acteurs contribuant à la création du logiciel, il est nécessaire d'utiliser des outils permettant d'automatiser la modification des modèles de caractéristiques afin d'éviter tout risque d'erreur [Reiser et Weber, 2007; Thein Tun et Patrick Heymans, 2009].

Thum *et al.* [2009] présentent une classification des différents changements qui peuvent être faits sur un modèle de caractéristiques :

- un *refactoring* consiste à modifier un modèle de caractéristiques en ayant toujours les mêmes configurations possibles,
- une *spécialisation* consiste à diminuer l'ensemble des configurations possibles,
- une *généralisation* consiste à rendre possibles de nouvelles configurations,
- une *édition arbitraire* est une modification qui ne correspond à aucune des catégories de changement énoncées précédemment, par exemple, une modification rendant impossible des configurations qui l'étaient et en créant de nouvelles.

Alves *et al.* [2006] cataloguent les différents types de refactoring sur un modèle de caractéristiques, par exemple, changer le niveau hiérarchique d'une caractéristique. Segura *et al.* [2008] identifient des règles à suivre pour fusionner deux modèles de caractéristiques.

Il existe plusieurs méthodes permettant d'automatiser la création d'un modèle de caractéristiques adapté à des besoins précis. Acher *et al.* [2010] proposent de composer un nouveau modèle de caractéristiques à partir de plusieurs. Ils ont créé pour cela différents opérateurs tels que *insert* et *merge*. Ainsi, il devient possible de créer un modèle de caractéristiques à partir de modèles fournis par des parties prenantes différentes.

Le choix des caractéristiques peut être effectué en fonction d'éléments extérieurs au logiciel. Acher *et al.* [2009] proposent de représenter les caractéristiques du contexte sous la forme d'un modèle de caractéristiques. Un ensemble de règles d'adaptation de type ECA

(Event / Condition / Action) permettent de définir quelles caractéristique du logiciel doivent être présentes ou absentes selon les caractéristiques du contexte. Par exemple, dans le cadre d'un logiciel de surveillance vidéo, des composants logiciels de traitement d'image différents sont requis en fonction du niveau de luminosité extérieur. Ainsi, le choix des caractéristiques peut être automatisé en fonction du contexte.

Fernandes *et al.* [2008, 2011] représentent le contexte d'exécution d'un futur produit à l'aide de deux nouveaux types de caractéristiques :

- les *entités du contexte*, par exemple, un utilisateur du système,
- les *informations du contexte*, par exemple, la position géographique d'un utilisateur du système.

Leur approche permet de créer un modèle de caractéristiques contenant à la fois les caractéristiques du logiciel et les caractéristiques de son contexte. Cependant, ils recommandent de faire deux modèles de caractéristiques distincts. L'un contenant les caractéristiques du logiciel, et l'autre celles du contexte. Les deux modèles étant liés par des règles restreignant la sélection des caractéristiques du logiciel en fonction de celles du contexte. Le principe suivi par cette approche est assez semblable à celui de Acher *et al.* [2009].

Jaroucheh *et al.* [2010b,a] cherchent à découpler le modèle de caractéristiques, et le contexte, représenté à l'aide d'une ontologie. Ils introduisent un système de *context proxy components* associés à des concepts d'un ontologie. Leur rôle est de communiquer, sous la forme d'un modèle de caractéristiques, l'état du contexte à un répertoire de services dont le rôle est de reconfigurer un logiciel en fonction du contexte.

3.4 Conclusion

Nous avons décrit dans ce chapitre le fonctionnement des lignes de produits logiciels, et plus particulièrement, les usages liés aux modèles de caractéristiques. Il existe un nombre important de méta-modèles de caractéristiques, et de méthodologies pour mettre en œuvre des lignes de produits logiciels. Malgré la standardisation en cours du langage CVL par l'OMG, la mise en œuvre de modèles de caractéristiques dépend des problématiques rencontrées dans les contextes spécifiques de chaque utilisation.

Il manque un méta-modèle de caractéristiques qui regroupe les concepts utiles dans un projet industriel (par exemple, les parties prenantes), et les concepts utiles à la création de logiciels adaptés à leur contexte (par exemple, multiplicités, représentation du contexte). De plus, il manque une méthode pour adapter automatiquement un modèle de caractéristiques au contexte d'exécution d'un futur produit. En effet, plusieurs approches proposent des solutions pour choisir automatiquement des caractéristiques en fonction d'un contexte, mais elles ne permettent pas de réaliser automatiquement la spécialisation d'un modèle de caractéristiques à un contexte donné. Dans ce cadre, il faudrait également permettre une traçabilité entre les éléments du contexte ayant eu un impact sur les changements apportés à un modèle de caractéristiques. Ce dernier point permettrait de fournir des informations

sur les objets du contexte à gérer aux composants logiciels déployés. Par exemple, cela permettrait de paramétrer automatiquement chaque composant logiciel d'optimisation uniquement pour les zones d'un bâtiment où les caractéristiques implémentées par ces composants ont été sélectionnées.

Les chapitres 4 et 5 proposeront des réponses à ces problématiques. Dans le chapitre 4, nous proposons un méta-modèle de caractéristiques réalisé à partir d'une synthèse des méta-modèles existants. Notre nouveau méta-modèle satisfait les besoins que nous avons identifiés dans le cadre d'un projet industriel et, plus particulièrement, dans le cadre d'un logiciel d'optimisation énergétique de bâtiments. Nous définissons également un profil UML permettant d'identifier dans un modèle métier, dont l'instance représente le contexte d'exécution d'un futur produit, les classes dont les instances influencent la spécialisation du modèle de caractéristiques.

Le chapitre 5 reprend les concepts clefs introduits dans notre nouveau méta-modèle à l'aide d'une notation mathématique et présente un algorithme générique permettant d'adapter un modèle de caractéristiques à un contexte donné.

CHAPITRE

4 Un nouveau méta-modèle de caractéristiques pour la prise en compte automatisée du contexte, et son implémentation

Sommaire

4.1 Introduction **39**
4.2 Synthèse des concepts décrits par les méta-modèles de caractéristiques existants **40**
4.3 Méta-modèle proposé **48**
4.4 Outillage du méta-modèle proposé **55**
4.5 Comparaison avec les outils existants **65**
4.6 Conclusion **65**

4.1 Introduction

Nous avons présenté dans le chapitre 2 la définition d'un exemple de modèle métier en UML, dans le domaine de la supervision de bâtiment, et indiqué que, dans un contexte industriel, le modèle de caractéristiques est destiné à être édité par des non spécialistes des lignes de produits. Dans ce chapitre, nous décrivons une synthèse des méta-modèles existants et des nouveaux concepts permettant de prendre en compte les spécificités de l'environnement d'exécution de chaque produit. À la suite de cela, nous proposons un nouveau méta-modèle de caractéristiques.

Nous avons donc procédé en deux étapes. Nous avons tout d'abord créé notre nouveau méta-modèle, puis nous avons étendu le langage UML avec les concepts spécifiques aux modèles de caractéristiques en créant un profil.

Dans la section 4.2, nous présentons les méta-modèles existants en identifiant les concepts clefs à inclure dans notre méta-modèle. La section 4.3 décrit notre méta-modèle et, plus particulièrement, les nouveaux concepts que nous avons développés afin de prendre

en compte le contexte d'exécution des produits. La section 4.4 décrit les étapes de la création du profil UML à partir du méta-modèle précédemment défini, et comment nous nous en sommes servi pour étendre les fonctionnalités du logiciel de modélisation UML Rational Software Architect (RSA). La section 4.5 compare notre approche avec des travaux présentant des similitudes. La section 4.6 conclut ce chapitre et propose quelques perspectives de développement futurs.

4.2 Synthèse des concepts décrits par les méta-modèles de caractéristiques existants

Cette section synthétise les méta-modèles de caractéristiques existants, et identifie les concepts les plus utiles selon les spécificités d'un projet industriel telles qu'énumérées au chapitre 2.

Nous analysons les méta-modèles existants selon les critères suivants :

1. *Capacité à définir des caractéristiques [4.2.1] :*
   Ce critère permet de comparer les définitions de caractéristiques et quels types d'artefacts elles peuvent représenter.
2. *Classification des relations entre caractéristiques [4.2.2] :*
   Nous subdivisons en quatre catégories distinctes les différents types de relations liant les caractéristiques.
   a) *Description de relations hiérarchiques [4.2.2] :*
      Ce critère vise à analyser la sémantique des relations hiérarchiques entre caractéristiques.
   b) *Définition des contraintes entre caractéristiques [4.2.2] :*
      Ce critère concerne la façon d'exprimer les contraintes de choix de caractéristiques, telles que des dépendances ou des exclusions.
   c) *Capacité à contraindre la sélection de sous-caractéristiques [4.2.2] :*
      Ce critère définit les contraintes qui imposent le choix de caractéristiques parmi un groupe.
   d) *Moyens d'identifier les caractéristiques obligatoires et optionnelles [4.2.2] :*
      Ce critère décrit de quelle façon identifier les caractéristiques obligatoires ou optionnelles, par exemple, à l'aide d'une multiplicité ou de contraintes.
3. *Groupements de caractéristiques [4.2.3] :*
   Ce critère décrit comment les caractéristiques peuvent être groupées conformément à un domaine métier, ou à des niveaux d'abstractions.
4. *Informations liées aux produits et aux configurations [4.2.4] :*
   Ce critère détermine quels types d'informations peuvent aider à implémenter un produit à partir d'un choix de caractéristiques.

4.2.1 Définition du concept de *caractéristique*, ou *feature*

Le concept de *caractéristique*, ou *feature*, a été initialement défini par Kang *et al.* [1990] avec la méthode *Feature-Oriented Domain Analysis* (FODA). Une *feature* y est définie comme étant une caractéristique d'une application décrite avec le vocabulaire du domaine. Par la suite, différentes contributions ont enrichi cette définition.

- La méthode FORM [Kang *et al.*, 1998] propose de classer les caractéristiques en quatre catégories :
  - *Capacités du logiciel*, par exemple services, opérations, caractéristiques non fonctionnelles, et niveaux de qualité de service,
  - *Environnement opérationnel*, par exemple système d'exploitation, logiciels (par exemple DB2),
  - *Techniques liées au domaine*, par exemple, processus métier, standards, législation,
  - *Implémentation des techniques du domaine*, par exemple protocole réseau, algorithmes, choix de conception.
- Fey *et al.* [2002] ajoutent le concept de *pseudo feature* représentant une caractéristique abstraite ne pouvant être sélectionnée qu'à la condition d'intégrer une ou plusieurs de ses sous-caractéristiques concrètes. La figure 4.4 de la section suivante présente un exemple d'utilisation d'une *pseudo feature* avec les relations introduites par leur approche.
- Zhang *et al.* [2006, 2004] définissent les features comme étant un ensemble cohérent de besoins représentant des caractéristiques visibles d'un système logiciel.
- Czarnecki *et al.* [2005, 2004a] considèrent que les caractéristiques sont des propriétés du logiciel pertinentes pour certains acteurs. Ils proposent d'associer une multiplicité à toute caractéristique n'appartenant pas à un groupe. La multiplicité spécifie combien de fois la caractéristique et ses sous-caractéristiques peuvent être dupliquées. Par exemple, la figure 4.1 présente un diagramme de caractéristiques dans lequel la caractéristique C possède la multiplicité *(0, *)*. Ainsi, la caractéristique C peut ne pas être incluse dans une configuration, ou être dupliquée autant de fois que nécessaire. Le cercle noir au dessus de la caractéristique B signifie que la caractéristique est obligatoire. C'est équivalent à une multiplicité *(1, n)* avec $n \in \mathbb{N}$ ou *(1, *)*.

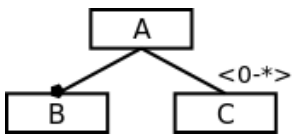

FIGURE 4.1 : Exemple de diagramme de caractéristiques selon Czarnecki *et al.* [2005, 2004a]

- Riebisch [2003] proposent quatre catégories de caractéristiques :
  1. les *caractéristiques fonctionnelles* concernent directement l'utilisateur,
  2. les *caractéristiques d'interface* assurent la conformité du produit final à des standards permettant son interopérabilité avec d'autres logiciels ou périphériques,
  3. les *caractéristiques de paramétrage* sont associées à une valeur qualitative ou

quantitative, telles que la mémoire nécessaire au bon fonctionnement d'un logiciel ou le temps de réponse attendu,

4. les *caractéristiques conceptuelles* ne possèdent pas d'implémentation (de façon similaire aux *pseudo-features* de Fey *et al.* [2002]). Ce sont ces caractéristiques abstraites dont les sous-caractéristiques appartiennent à l'une des catégories précédentes. La racine du modèle de caractéristiques est toujours elle-même une caractéristique conceptuelle.

- Zhang *et al.* [2004] présentent le concept de *binding-time* afin de définir à quelle phase du cycle de développement du logiciel une caractéristique donnée doit intervenir. Par exemple, au moment du choix de réutilisation d'un composant logiciel, ou de la compilation, ou de l'installation du logiciel, ou du démarrage du logiciel.
- Fey *et al.* [2002] ont introduit le concept de *propriété*, similaire au concept d'*attribut* proposé par Czarnecki *et al.* [2005, 2004a]. Cela permet d'associer à une caractéristique une information supplémentaire la concernant (par exemple, la quantité de mémoire relative à une caractéristique de stockage).

En résumé, la définition proposée par Czarnecki *et al.* [2005, 2004a] permet de décrire les caractéristiques d'une façon plus adaptée aux méthodes de construction de logiciels grâce à l'association d'une multiplicité. Sa combinaison avec les perspectives de *FODA* nous permet d'organiser un modèle de caractéristiques en fonction des nombreux domaines métiers impliqués dans un projet industriel.

4.2.2 Relations entre caractéristiques

Relations hiérarchiques

Différentes sortes de liens hiérarchiques, de spécialisation, ou de composition ont été proposés. Le tableau 4.1 illustre les différentes variations relatives à ces liens.

TABLE 4.1 : Relations hiérarchiques

	Décomposition	Spécialisation	
		Enrichissement	Réalisation
FODA [Kang *et al.*, 1990]	Consists-of	N.A. [1]	N.A.
FORM [Kang *et al.*, 2002]	Composed-of	Generalization Specialization	Implemented-by
Fey *et al.* [2002]	Refine	N.A.	Provided-by
Zhang *et al.* [2004]	Decompose Detail	Specialize	N.A.
Czarnecki *et al.* [2005]	Relation	N.A.	N.A.

- La relation *Consists-of* de la méthode FODA [Bontemps *et al.*, 2004; Kang *et al.*, 1990; Schobbens *et al.*, 2007] représente la décomposition d'une caractéristique en plusieurs "sous-caractéristiques". Par exemple, une caractéristique *Extract Transform Load* (ETL), appartenant à une ligne de produits contenant un data warehouse,

1. Non Applicable

consiste en trois sous-caractéristiques : *Extraire des données*, *Transformer les données*, et *Charger les données*. La figure 4.2 présente les caractéristiques d'un ETL.

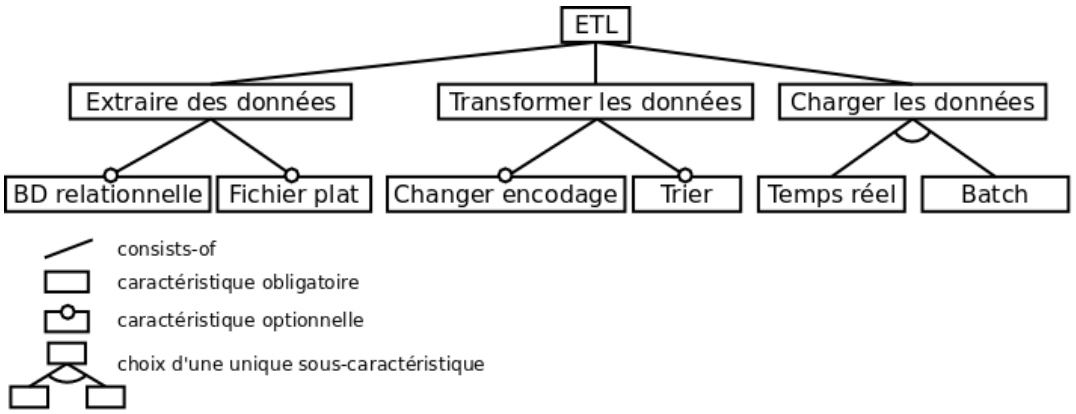

FIGURE 4.2 : Exemple montrant les caractéristiques d'un ETL selon la représentation de la méthode FODA

- La méthode *FORM* [Kang *et al.*, 1998, 2002] introduit les trois relations suivantes :
  - *Composed-of*, pour décrire les constituants d'une caractéristique,
  - *Generalization / Specialization*, pour spécialiser ou généraliser une caractéristique,
  - *Implemented-by*, pour décrire le fait qu'une caractéristique de haut niveau peut être implémentée par une sous-caractéristique.

  Comme le montre la figure 4.3 avec un exemple issu du même domaine que celui de la figure 4.2, la sémantique de cette méthode améliore la compréhension du diagramme. Les caractéristiques *DB2* et *PostgreSQL* implémentent la caractéristique *Base de données*. Les caractéristiques *Temps réel* et *Batch* spécialisent la caractéristique *Charger les données*. Les caractéristiques *Extraire des données*, *Transformer les données*, et *Charger les données* composent la caractéristique *ETL*.

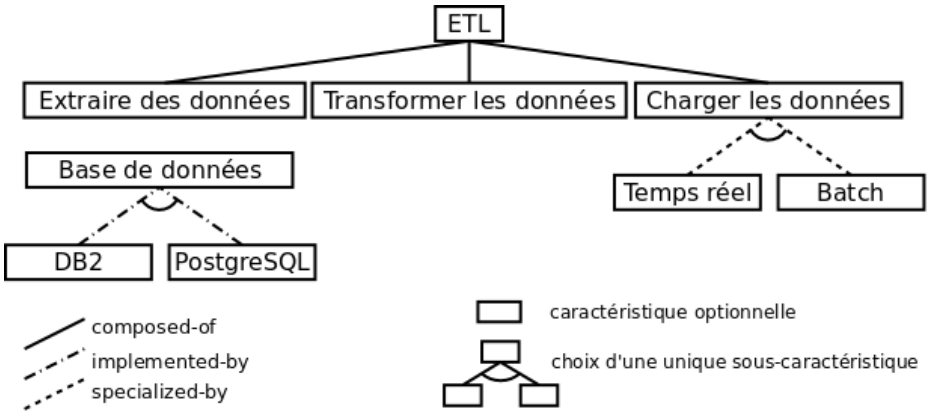

FIGURE 4.3 : Exemple montrant les caractéristiques d'un ETL selon la représentation de la méthode FORM

- Fey *et al.* [2002] utilisent deux types de liens hiérarchiques entre caractéristiques :

– La relation *refine* affine la caractéristique parent en une caractéristique ayant un niveau d'abstraction inférieur. Ils considèrent cette relation similaire à la relation *consists-of* introduite par la méthode FODA.
– La relation *provided-by* lie une pseudo-feature aux caractéristiques qui la réalisent. Cette relation est similaire à la relation *implemented-by* de la méthode FORM.

De plus, leur proposition permet la construction de modèles de caractéristiques sous forme de graphes orientés sans circuits en permettant à une caractéristique d'avoir plusieurs parents auxquels elle est associée par une relation *refine*. Comme décrit dans la section précédente, une *pseudo feature* est "implémentée" par des caractéristiques associées par la relation *provided-by*. La figure 4.4 montre les caractéristiques d'un système de supervision de bâtiments. La caractéristique *Droit d'accès visiteurs* est implicitement héritée par les caractéristiques *Voir consignes* et *Modifier consignes*. Ainsi, ces deux caractéristiques fournissent l'implémentation de la caractéristique *Supervision de bâtiments*. Le choix de l'une de ces deux caractéristiques implique pour chacune l'utilisation des caractéristiques associées par la relation *refine* à *Supervision de bâtiments*.

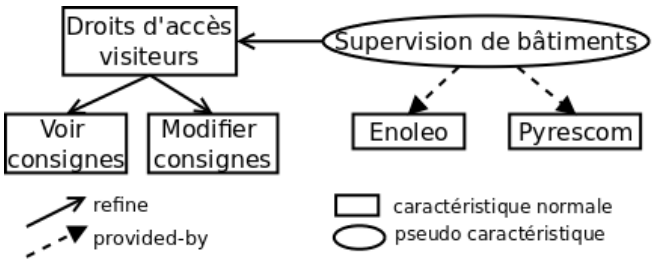

FIGURE 4.4 : Exemple montrant les caractéristiques d'un système de supervision de bâtiments avec la méthode de Fey *et al.* [2002]

- Zhang *et al.* [2004] identifient trois types de relations hiérarchiques :
  1. *decomposition* : permet de décomposer une caractéristique en sous-caractéristiques. La caractéristique ne peut être sélectionnée que si toutes ses sous-caractéristiques le sont également. Par exemple, une caractéristique *Édition* dans un logiciel de traitement de texte, est décomposée en sous-caractéristiques *Copier*, *Coller*, et *Couper*.
  2. *detailization* : permet de détailler les fonctionnalités d'une caractéristique. La caractéristique peut être sélectionnée même si ses sous-caractéristiques ne le sont pas. Par exemple, une caractéristique *Dashboard consommation énergétique bâtiment* peut être détaillée avec une caractéristique *Affichage plan bâtiment 3D*.
  3. *specialization* : permet à une sous-caractéristique d'ajouter des fonctionnalités à la caractéristique parent. Par exemple, une caractéristique *Affichage plan bâtiment 3D* peut être spécialisée par une caractéristique *Affichage 3D des consommations énergétiques du bâtiment*.
- Czarnecki *et al.* [2005, 2004a,b] ont décidé de ne pas prendre en compte ces différents types de relations et utilisent de simples relations "hiérarchiques" afin de simplifier.

La tableau 4.1 présente trois différentes catégories de relations hiérarchiques regroupant les types de relations présentés précédemment.

Nous avons choisi de déduire de ces trois catégories les relations hiérarchiques que nous intégrons dans notre méta-modèle :

- *decomposition*, permettant de détailler une caractéristique,
- *specialization*, avec les relations :
  - *enrichment*, permettant aux sous-caractéristiques de décrire des fonctionnalités enrichissant celles de la caractéristique parent,
  - *realization*, ou *implementation*, décrivant différents choix d'implémentation.

Contraintes entre caractéristiques

Les contraintes servent à éliminer les caractéristiques inutiles ou incompatibles au sein d'une configuration. Ces contraintes sont définies en fonction du domaine décrit par le modèle de caractéristiques.

La méthode *FODA* utilise des règles de composition pour décrire la façon dont les caractéristiques dépendent les unes des autres dans une configuration : une caractéristique peut nécessiter la présence d'une autre, ou celles-ci peuvent s'exclure. Ces règles sont décrites textuellement avec une syntaxe décrite dans [Kang *et al.*, 1990]. Riebisch [2003] introduis la notion de *recommandation* (*hint*) afin de recommander des caractéristiques à garder dans une configuration. Par exemple, la caractéristique *IBM Infosphere DataStage*, décrivant un ETL, recommande la caractéristique *IBM DB2 data warehouse* car les logiciels représentés par ces caractéristiques ont été optimisés pour fonctionner ensemble.

Nous avons gardé les deux contraintes de FODA ainsi que les recommandations.

Contraintes sur la sélection d'un groupe de sous-caractéristiques

- Les méthodes *FODA* et *FORM* proposent deux façons de représenter la sélection de sous-caractéristiques. Une simple association entre deux caractéristiques signifie qu'il n'y a pas de contraintes de choix. Un arc de cercle dessiné à travers plusieurs associations signifie que les sous-caractéristiques sont *groupées* et qu'une seule d'entre elles peut être sélectionnée (les sous-caractéristiques s'excluent mutuellement). Une caractéristique est obligatoire par défaut, mais la présence d'un cercle au dessus du nom signifie qu'elle est optionnelle. La figure 4.5 présente un exemple de modèle de caractéristiques selon cette représentation graphique.
- De même, Fey *et al.* [2002] expriment l'absence de contraintes de choix avec une simple association. Un ensemble de sous-caractéristiques s'excluant mutuellement est décrit avec des contraintes d'exclusion mutuelle entre toutes les caractéristiques concernées. Une caractéristique obligatoire est décrite avec une contrainte "requiert" issue de sa caractéristique parent. Dans l'exemple de la figure 4.6, E est optionnelle

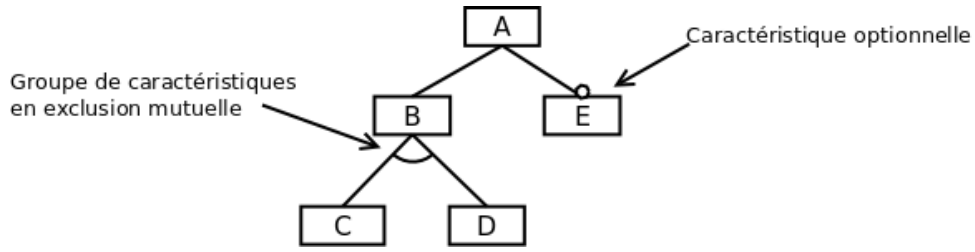

FIGURE 4.5 : Exemple de diagramme de caractéristiques selon la représentation de la méthode FODA

car elle n'est pas associée à A par une relation "requiert" (représentée par une flèche en pointillé dans la syntaxe concrète de Fey *et al.* [2002]).

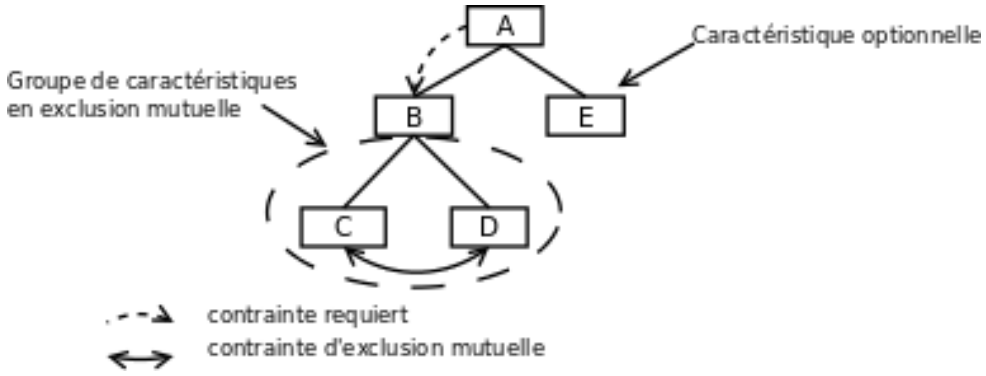

FIGURE 4.6 : Exemple de diagramme de caractéristiques selon la représentation de Fey *et al.* [2002]

- Riebisch [2003] et Czarnecki *et al.* [2005] utilisent des multiplicités pour définir combien de caractéristiques groupées peuvent être choisies. Le groupe est également symbolisé par un arc de cercle dessiné sur les associations entre la caractéristique parent et ses sous-caractéristiques. Par exemple, la multiplicité *(1, 1)* signifie qu'une et seulement une sous-caractéristique doit être choisie. La multiplicité *(0, 1)* signifie qu'aucune ou au maximum une sous-caractéristique peut être choisie.

En résumé, la façon la plus complète de contraindre le choix dans un groupe de caractéristiques est de spécifier à l'aide d'une multiplicité le nombre de caractéristiques pouvant être choisies. Cependant, dans un souci de clarté de présentation aux non spécialistes, nous avons choisi de garder un nom décrivant le type de choix : *or, and* et *xor*.

Identification des caractéristiques obligatoires et optionnelles

- Dans les modèles de caractéristiques utilisés par les méthodes *FODA* et *FORM*, toutes les caractéristiques sont obligatoires par défaut. Les caractéristiques optionnelles sont représentées graphiquement à l'aide d'un petit cercle, comme décrit par la figure 4.5.

- Pour Fey *et al.* [2002], toutes les caractéristiques sont optionnelles. Une contrainte *require* entre une caractéristique et sa sous-caractéristique décrit une sous-caractéristique comme étant obligatoire. L'absence de contrainte décrit une sous-caractéristique optionnelle.
- Czarnecki *et al.* [2004a] utilisent des multiplicités pour exprimer l'optionalité d'une caractéristique. Par exemple, une multiplicité *(1,1)* décrit une caractéristique obligatoire, et une multiplicité *(0,1)* décrit une caractéristique optionnelle. Lorsque la multiplicité maximale est supérieure à 1, elle spécifie combien de fois la caractéristique peut être dupliquée au sein d'une configuration.

Nous avons choisi d'utiliser les multiplicités pour exprimer l'optionalité d'une caractéristique. Cela nous permet de spécifier combien de fois une caractéristique peut être dupliquée dans une même configuration.

4.2.3 Groupements arbitraires de caractéristiques

Cette sous-section identifie comment les caractéristiques peuvent être groupées pour mieux illustrer des concepts spécifiques au domaine du modèle de caractéristiques.

- Fey *et al.* [2002] introduisent les *feature-sets*, ensemble de caractéristiques, afin de les grouper arbitrairement. Ils servent à représenter des vues spécifiques à chaque partie prenante [2] (stakeholder) contenant des caractéristiques qui leur sont propres.
- Czarnecki *et al.* [2004a, 2005] permettent à des caractéristiques de référencer d'autres modèles de caractéristiques. Cela évite de les dupliquer dans le cas où ils sont présents dans différents composants. Par exemple, des caractéristiques de contrôle d'accès peuvent être présentes dans plusieurs composants d'un même logiciel. Les caractéristiques représentant ces composants peuvent donc référencer un modèle de caractéristiques du sous système de contrôle d'accès, et donc éviter une représentation redondante.
- Fey *et al.* [2002] proposent également de créer des contraintes entre plusieurs groupes. Par exemple, ils permettent de contraindre la sélection d'une caractéristique si au moins l'une des caractéristiques d'un groupe est sélectionnée. Une possible application de nos groupements arbitraires de caractéristiques est la modélisation des différents critères permettant d'atteindre un niveau donné de satisfaction client [Noriaki Kano *et al.*, 1984].
  Les caractéristiques relatives à chaque niveau de satisfaction peuvent être groupées à l'aide de *feature-sets* :
  – *basic requirements* : caractéristiques indispensables aux utilisateurs du produit,
  – *satisfiers* : caractéristiques apportant une plus grande satisfaction aux utilisateurs,
  – *delighters* : caractéristiques accessoires, mais apportant plus de satisfaction,
  – *indifferent requirements* : caractéristiques n'apportant aucune valeur ajoutée aux utilisateurs.

2. Une partie prenante représente une personne (par exemple, un expert du domaine, un architecte informatique, ...) autorisée à choisir des caractéristiques d'un nouveau produit.

Nous avons choisi d'adapter le concept de *niveaux*, ou *layers*, introduit dans la méthode FORM. Cela permet d'organiser les caractéristiques selon leur nature (par exemple, les caractéristiques fonctionnelles, non fonctionnelles, liées aux choix d'implémentation).

Nous proposons aussi d'adapter les *feature-sets* de Fey *et al.* [2002] en utilisant le méta-modèle de contraintes de Zhang *et al.* [2004]. Cela permet de décrire des contraintes de sélection de caractéristiques au sein d'un même groupe. Par exemple, cela permet d'obliger la sélection de l'ensemble des caractéristiques d'un groupe défini ou aucune, ou d'une seule.

4.2.4 Informations liées aux produits et aux configurations

Zhang *et al.* [2004] utilisent des attributs de caractéristiques pour représenter leur *binding-state*. Ils déterminent à quelle phase du cycle de développement une caractéristique peut être choisie.

De plus, le modèle de caractéristiques peut inclure des règles logiques pour décrire les contraintes les liant. Cela permet d'exprimer des contraintes de façon équivalente ou plus précise qu'avec de simples associations représentées graphiquement. Nous n'avons pas créé de concept dédié aux contraintes dans le méta-modèle et le profil car le langage UML permet, de façon native avec le langage OCL, de décrire des contraintes logiques entre les éléments d'un modèle.

4.3 Méta-modèle proposé

Dans cette section nous présentons notre méta-modèle, qui est le résultat de l'analyse présentée en 4.2.

La figure 4.7 présente sous la forme d'un tableau contenant les principales classes qui composent notre méta-modèle de caractéristiques. Une instance de la classe *ProductLine* est composée d'instances des classes présentées dans la colonne de gauche. Chaque cellule de la colonne de droite décrit les propriétés de la cellule correspondante à gauche.

Afin d'illustrer la définition du méta-modèle de cette section, l'annexe A.1 décrit la grammaire du langage permettant de construire textuellement des modèles de caractéristiques.

- *ProductLine* représente la ligne de produits,
- *Feature* représente les caractéristiques d'une ligne de produits,
- *Product* représente les configurations de produits.

Une ligne de produits contient des caractéristiques et a des configurations de produits. Une configuration de produit est composée d'un ensemble de caractéristiques de la ligne de produits. L'ensemble des caractéristiques choisies dans un produit doit satisfaire :

- les contraintes représentées ici par les associations *requiredFeature* lorsqu'une carac-

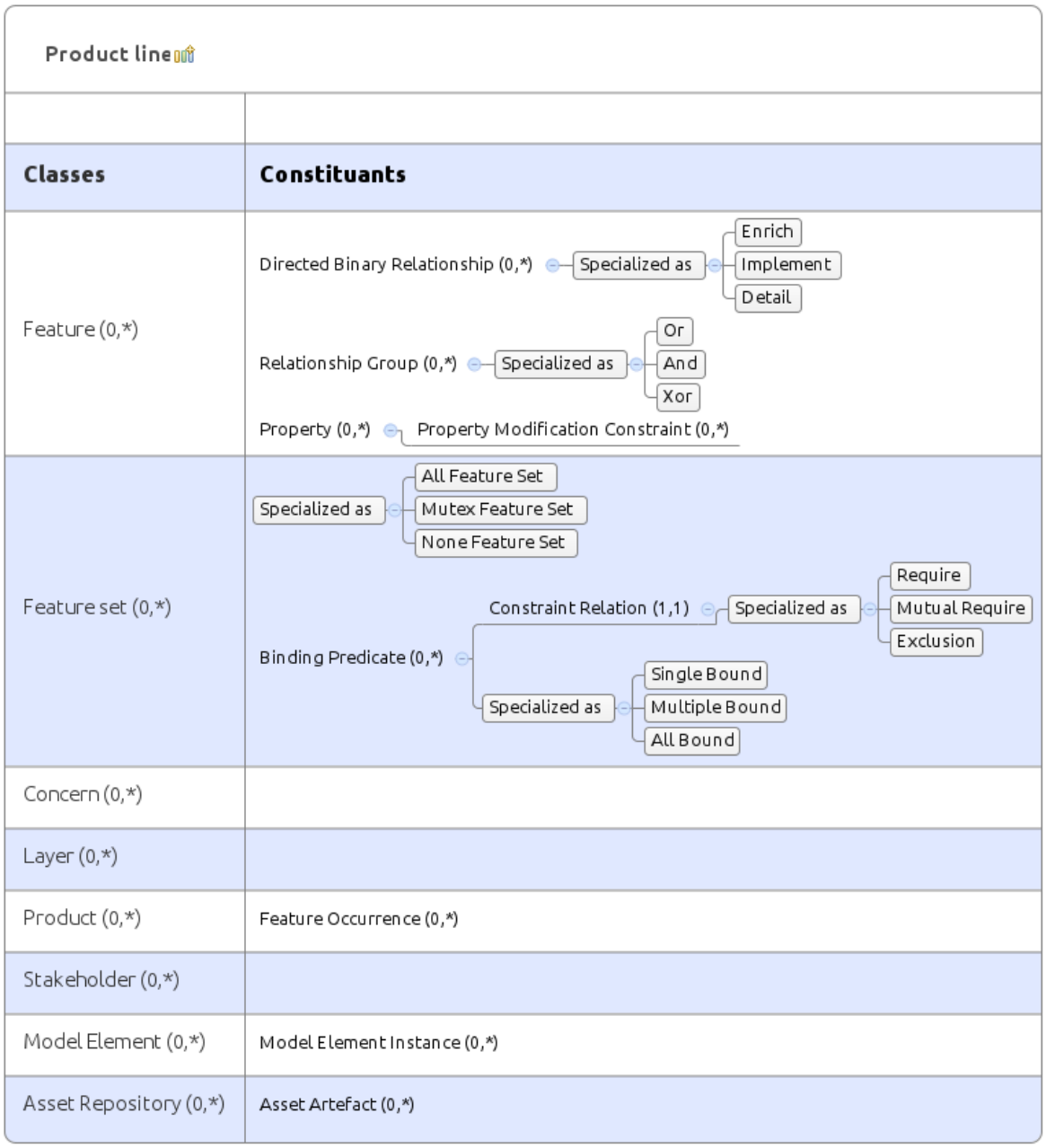

FIGURE 4.7 : Tableau résumant les concepts clefs de notre méta-modèle de caractéristiques

téristique nécessite la présence d'une autre, et *conflictingFeature* lorsque deux caractéristiques sont en exclusion mutuelle,
- les contraintes relatives aux *feature-sets*,
- les contraintes liées aux groupes de relations hiérarchiques entre caractéristiques,
- les contraintes exprimées sous forme d'expressions logiques.

La figure 4.8 présente un extrait du méta-modèle concernant les propriétés de caracté-

ristiques. Les propriétés ont un *type de variabilité*, spécifié par l'énumération *VariabilityKind* pour déterminer à quel moment la valeur d'une propriété peut être modifiée :

- *fixed* : La valeur de la propriété est fixée pour tous les produits de la ligne de produits.
- *variable* : La valeur de la propriété dépend des propriétés des autres caractéristiques ayant été choisies.
  Par exemple, la mémoire tampon du champ texte d'une interface graphique peut changer en fonction des informations devant être stockées (un nom ou une adresse).
- *family-variable* : Une propriété peut varier d'un produit à l'autre en fonction des caractéristiques choisies.
  Par exemple, la capacité d'un répertoire téléphonique dépend de la présence d'une extension de mémoire et de la capacité de la carte sim.
- *user-defined* : La valeur de la propriété peut être choisie librement au moment de la configuration.
  Par exemple, la valeur de la propriété spécifie la fréquence des sauvegardes d'un logiciel.

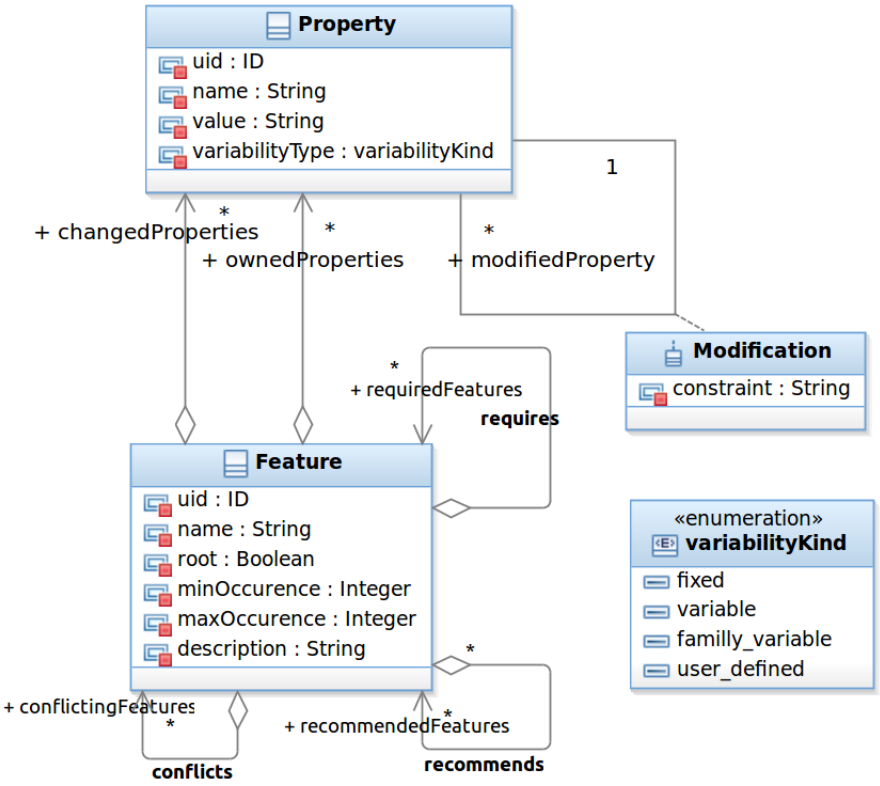

FIGURE 4.8 : Extrait du méta-modèle représentant les propriétés de caractéristiques

La figure 4.8 présente également les contraintes entre caractéristiques. Les contraintes d'exclusion mutuelle et d'exigence entre caractéristiques associent des caractéristiques indépendamment de leur position dans la hiérarchie du modèle. Elles sont modélisées respectivement par les associations *conflictingFeature* et *requiredFeature*. Nous avons ajouté

l'association *recommendedFeature* afin de pouvoir recommander des caractéristiques pertinentes, mais optionnelles, permettant d'améliorer le fonctionnement du produit.

Les relations hiérarchiques et les groupes de relations sont décrites dans la figure 4.9. Les relations entre une caractéristique et ses sous-caractéristiques sont groupées par la classe *RelationshipGroup*. Ses attributs décrivent la multiplicité permettant de restreindre le nombre de sous-caractéristiques pouvant être choisies. La multiplicité peut être choisie librement, sauf pour les trois spécialisations de cette classe. *OrGroup* requiert une multiplicité (0, *), *AndGroup* une multiplicité (*, *), et *XorGroup* une multiplicité (1, 1).

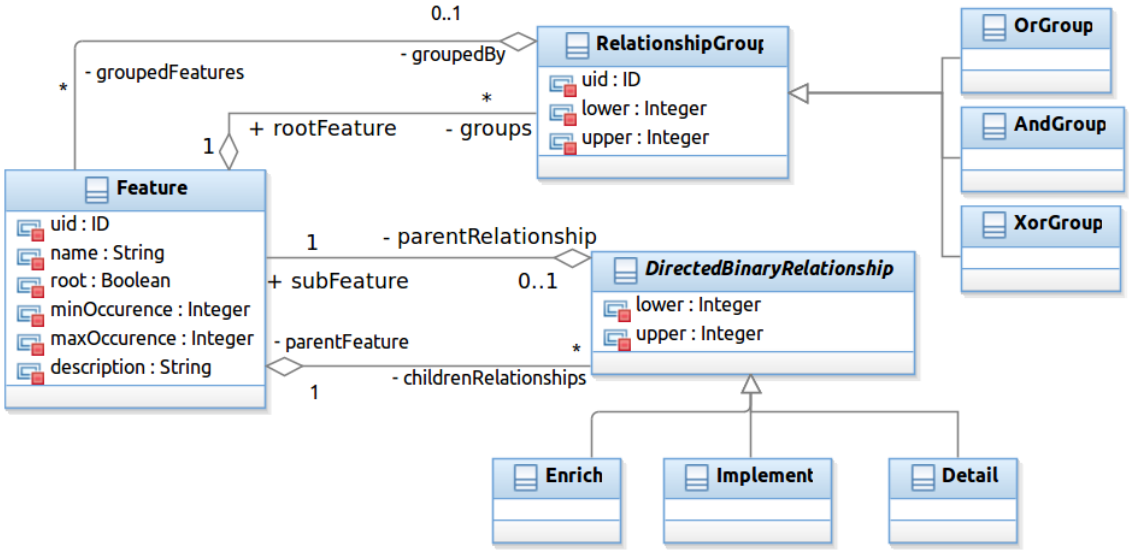

FIGURE 4.9 : Relations hiérarchiques et groupes de relations

La classe *DirectedBinaryRelationship* représente l'association qui lie une caractéristique à ses sous-caractéristiques. Elle est spécialisée par les classes *Enrich*, *Implement*, et *Detail*.

La figure 4.10 montre comment les niveaux (représentés par la classe *Layer*) et les ensembles de caractéristiques (représentés par la classe *FeatureSet*) peuvent être associés avec les parties prenantes (représentées par la classe *Stakeholder*) de la ligne de produits.

Les ensembles de caractéristiques et niveaux peuvent être associés à un centre d'intérêt (représenté par la classe *Concern*). Cela permet d'exprimer que les caractéristiques associées aux *feature-sets* et niveaux, eux-mêmes associés à un centre d'intérêt, doivent être choisies par les parties prenantes associées à ce centre d'intérêt.

La classe *Layer*, représentant un niveau, permet de grouper des caractéristiques. Une caractéristique ne peut appartenir qu'à un seul niveau à la fois. Les niveaux jouent un rôle similaire à celui des "packages" du langage Java.

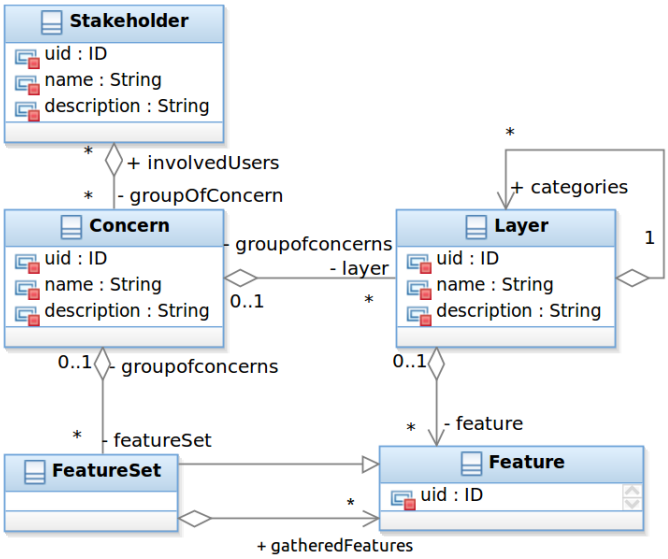

FIGURE 4.10 : Centres d'intérêts des parties prenantes

La classe *FeatureSet* hérite de la classe *Feature*. Cette spécialisation permet de grouper arbitrairement des caractéristiques, par exemple, par domaine métier, ou pour représenter les caractéristiques permettant d'être conforme à une norme.

La figure 4.11 présente l'application des contraintes aux *feature-sets* :

- *MutexFeatureSet* : une seule caractéristique peut être choisie dans le *feature-set*,
- *NoneFeatureSet* : il n'y a pas de contraintes entre les caractéristiques,
- *AllFeatureSet* : toutes les caractéristiques doivent être sélectionnées, ou aucune.

La classe *ConstraintRelation* décrit la relation entre deux *feature-sets*. Par exemple, un *feature-set* peut nécessiter la présence de caractéristiques d'un autre *feature-set*, ou deux *feature-sets* peuvent s'exclure mutuellement. Un *BindingPredicate* est utilisé pour représenter comment les contraintes doivent être appliquées sur chaque *feature-set*.

La figure 4.12 est un diagramme d'instances UML qui présente une contrainte entre deux feature sets. Dans cet exemple, la contrainte spécifie que si une caractéristique du feature set *fs1* est sélectionnée, alors toutes les caractéristiques du feature set *fs2* doivent également être sélectionnées. Plus précisément, l'objet *bp1* de type *SingleBound* spécifie que la contrainte porte sur une seule caractéristique de *fs1*. L'objet *bp2* de type *AllBound* spécifie que la contrainte s'applique à l'ensemble des caractéristiques de *fs2*. La contrainte *cr*, de type *Require*, spécifie que la sélection d'une caractéristique de *fs1* requiert la sélection de toutes les caractéristiques de *fs2*.

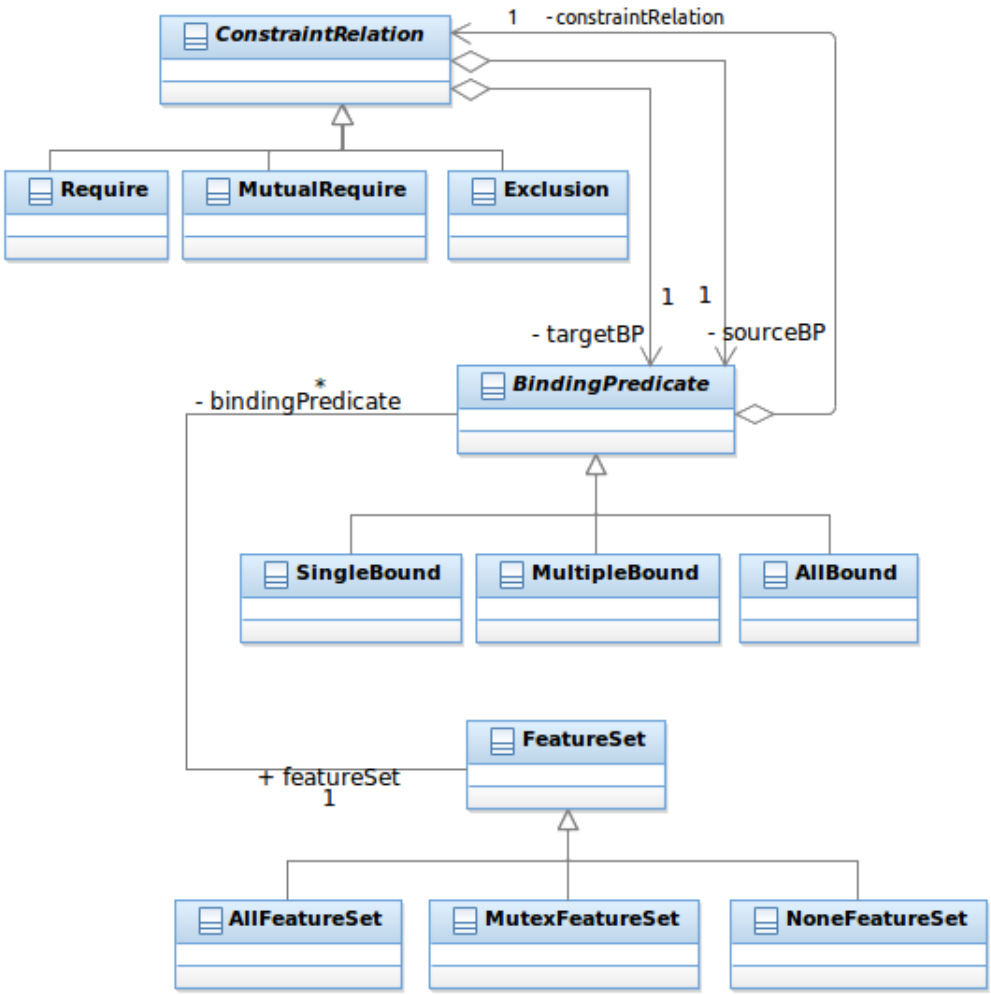

FIGURE 4.11 : Contraintes applicables aux feature sets

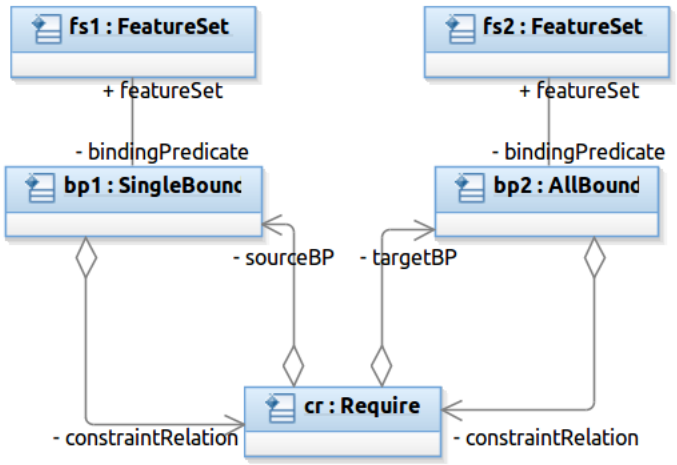

FIGURE 4.12 : Diagramme d'instances présentant une contrainte entre deux feature sets

Nous avons choisi d'utiliser la norme *Reusable Asset Specification* (RAS) [OMG, 2005] pour décrire le ou les fichiers, aussi appelés artefacts, permettant de mettre en œuvre dans un produit la fonctionnalité décrite par une caractéristique. La spécification de cette norme définit qu'un *asset* est composé d'artefacts. Un asset est géré et référencé par un dépôt d'assets. Un asset peut inclure, par exemple, des documents techniques, du code, des librairies logicielles. De façon plus générale, un asset permet de classifier tout élément produit durant le cycle de vie d'un logiciel.

La figure 4.13 décrit le lien entre caractéristiques et assets. Ainsi, une caractéristique peut être associée à plusieurs artefacts d'assets, et un artefact peut être représenté par plusieurs caractéristiques. De plus, nous avons modélisé le concept *AssetRepository* qui représente le dépôt où est stocké chaque asset.

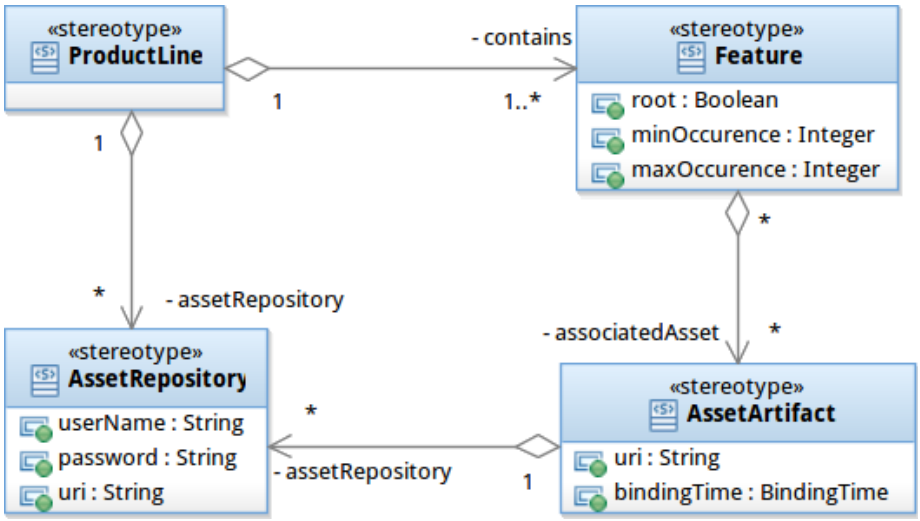

FIGURE 4.13 : Relations entre assets et caractéristiques

La figure 4.14 décrit les classes permettant de définir la configuration d'un produit. Un produit est composé d'occurrences de caractéristiques. Ainsi, la classe *Product* est associée à la classe *FeatureOccurence* représentant une occurrence de caractéristique dans un produit. Une occurrence de caractéristique est associée à une unique caractéristique (classe *Feature*) et zéro ou une instance de concept. La classe *ModelElement* décrit un concept métier traité par un produit, par exemple le concept *Bâtiment*. La classe *ModelElementInstance* décrit une instance de concept métier, par exemple un bâtiment particulier, le bâtiment *B2*.

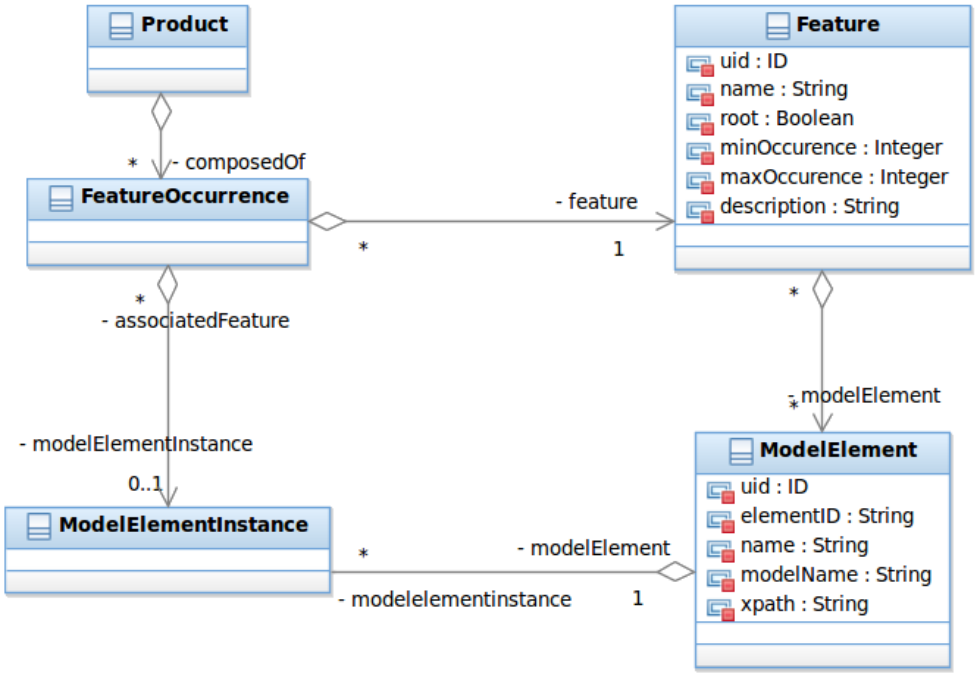

FIGURE 4.14 : Configuration d'un produit

4.4 Outillage du méta-modèle proposé

Dans la section 4.3, nous avons présenté notre méta-modèle de caractéristiques. Nous avons dû produire des outils pour l'expérimenter en créant une ligne de produits logiciels. Dans cette section nous décrivons nos choix et notre méthodologie pour mettre en œuvre un prototype d'outil de création de modèles de caractéristiques conformes à notre méta-modèle.

Plusieurs solutions nous permettraient de mettre en œuvre notre prototype :

1. créer un méta-modèle sous Eclipse, et utiliser le framework *Eclipse Modelling Framework* (EMF) pour générer son éditeur,
2. créer un langage conforme à une grammaire, exprimée par exemple avec EBNF, et générer un éditeur permettant de définir textuellement des modèles de caractéristiques,
3. créer un profil UML intégré à RSA et générer son éditeur.

La solution d'Eclipse est un environnement d'ingénierie des modèles complet mais assez lourd à mettre en œuvre. La création d'un langage conforme à une grammaire permet de construire rapidement et simplement un prototype d'outil de création de modèles de caractéristiques. Cependant, dans un contexte industriel, un langage textuel complique la présentation et la communication des diagrammes de caractéristiques. Notre choix doit favoriser un outil graphique pour faciliter à des non informaticiens la création d'un modèle et réduire le temps d'apprentissage du langage. La solution du profil UML intégré à RSA consiste à étendre le langage UML avec les concepts relatifs aux diagrammes de caractéristiques et à générer ensuite un outil permettant l'utilisation de ces nouveaux concepts dans

des diagrammes. La solution est assez rapide à mettre en œuvre. L'intégration de l'approche par ligne de produits dans un contexte industriel est simplifiée car RSA est un outil répandu.

Nous avons donc choisi de baser notre prototype sur un profil UML intégré dans RSA. Cependant, nous avons aussi réalisé la grammaire du langage correspondant à notre méta-modèle ; elle est présentée dans l'annexe A de ce mémoire. Son utilisation pourrait faire l'objet d'un développement futur afin d'offrir une alternative à RSA.

4.4.1 Profil UML

Pour présenter notre méta-modèle de caractéristiques, nous avons choisi d'utiliser un profil UML. Cette section décrit nos choix de méta-classes UML, et quelles informations nous avons décrites avec quels stéréotypes. Le tableau 4.2 présente, alphabétiquement classés, l'ensemble de nos stéréotypes et la méta-classe que chacun d'eux étend.

Le stéréotype *AssetArtifact* référence les fichiers nécessaires (assets) à la mise en œuvre d'une caractéristique dans un produit. Un asset appartient à un dépôt d'assets, représenté par le stéréotype *AssetRepository*. Ces deux stéréotypes étendent la méta-classe *Class*.

Le stéréotype *BindingPredicate* étend la méta-classe *Port*. Il représente le type de contraintes portant sur un groupe de caractéristiques (un *feature-set*, dont le stéréotype étend la méta-classe *Component*). Un port stéréotypé par *BindingPredicate* ne peut appartenir qu'à un composant stéréotypé par *FeatureSet*.

Les centres d'intérêts des parties prenantes sont modélisés par le stéréotype *Concern*. Le stéréotype doit être associé aux parties prenantes, et étend la méta-classe *Class*. Un centre d'intérêt porte sur les caractéristiques appartenant à des niveaux (*Layer*) ou des groupes de caractéristiques (*feature-sets*). Il doit donc être associé à au moins un niveau ou au moins un groupe de caractéristiques.

Le stéréotype *ConstraintRelation* étend la méta-classe *Association*. Son rôle est de représenter une contrainte entre deux *feature-sets* ou entre deux *BindingPredicates*.

La relation hiérarchique entre deux caractéristiques est décrite par le stéréotype abstrait *DirectedBinaryRelationship* qui étend la méta-classe *Association*. Une relation hiérarchique lie deux caractéristiques ou un groupe de relations hiérarchiques de caractéristiques (un port stéréotypé par un stéréotype héritant de *RelationshipGroup*) et une caractéristique. Ce stéréotype est spécialisé par les stéréotypes *Detail*, *Enrich* et *Implement*. Leurs rôles respectifs a été présenté dans la sous-section 4.2.2.

Pour les caractéristiques, nous avons créé le stéréotype *Feature* qui étend la méta-classe UML *Component* car une caractéristique peut être vue comme un composant logiciel de haut niveau. Il s'agit du concept le moins éloigné. De plus, c'est une méta-classe dont la sémantique des liens peut être enrichie au moyen notamment de ports et qui peut être classée au moyen de paquetages.

Stéréotype proposé	Sémantique	Méta-classe UML étendue
AssetArtifact	Identifie l'asset associé à une caractéristique	Class
AssetRepository	Identifie un dépôt d'assets	Class
BindingPredicate	Représente un type de contraintes entre feature-sets	Port
Concern	Représente un ensemble de caractéristiques pertinentes pour une partie prenante	Class
ConstraintRelation	Identifie une contrainte portant sur deux *feature-sets*	Association
DirectedBinaryRelationship	Stéréotype abstrait représentant une relation hierarchique entre caractéristiques	Association
Feature	Représente une caractéristique	Component
Layer	Groupe un ensemble de caractéristiques	Package
ModelRelationship	Associe une caractéristique à un concept du contexte	Association
Modification	Représente l'impact d'une propriété sur une autre	Usage
ProductLine	Représente la ligne de produit, l'ensemble des caractéristiques et produits existants	Component
Property	Représente une propriété de caractéristique	Port
RelationshipGroup	Groupe un ensemble de relations hiérarchiques entre caractéristiques	Port
Stakeholder	Représente une personne impliquée dans la configuration d'un produit	Actor

TABLE 4.2 : Vue d'ensemble des stéréotypes

Un niveau, ou *Layer*, étend la méta-classe *Package* afin de permettre le groupement hiérarchique de caractéristiques.

Le stéréotype *ModelRelationship* étend la méta-classe *Association*. Il représente la dépendance d'une caractéristique avec un élément du méta-modèle métier, *i.e.*, une classe UML issue d'un autre modèle UML.

Le stéréotype *Modification* étend la méta-classe *Usage*. Le stéréotype définit un attribut destiné à contenir la description de la modification à effectuer sur la valeur de la propriété cible de la relation.

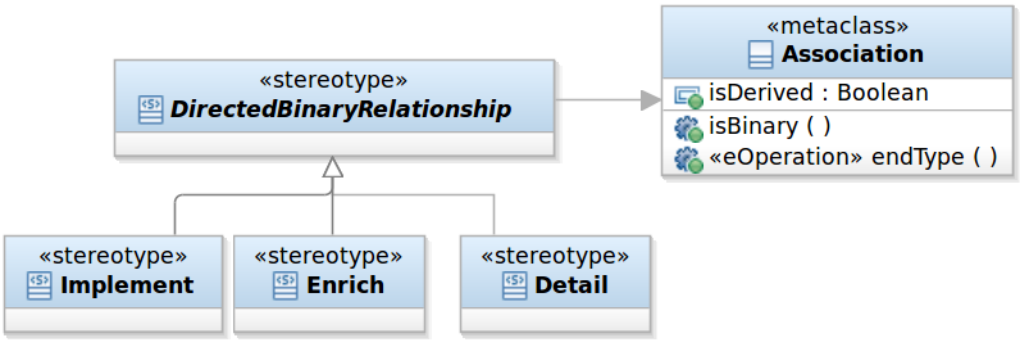

FIGURE 4.15 : Stéréotype *DirectedBinaryRelationship*

Le stéréotype *ProductLine* représente l'élément de base de tout modèle de caractéristiques. Il est associé à l'ensemble des caractéristiques et des configurations existantes. Il étend la méta-classe *Component*.

Le stéréotype *Property* étend la méta-classe *Port*. Un port représente un point d'interaction entre un *Classifier* et son environnement [OMG, 2011]. Nous étendons ce concept pour définir le concept de propriété de caractéristique dont la valeur peut être modifiée en fonction du choix d'autres caractéristiques.

Un groupe de relations, défini par le stéréotype *RelationshipGroup*, étend la méta-classe *Port* comme présenté dans la figure 4.16. Nous avons également créé, pour des raisons pratiques, les stéréotypes *Xor*, *Or* et *And* qui représentent des groupes prédéfinis avec les multiplicités *(1, 1)*, *(1, *)* et *(*,*)*. Le port stéréotypé doit se placer sur la caractéristique parent à partir de laquelle les relations sont groupées.

Le stéréotype *Stakeholder*, pour les parties prenantes, étend la méta-classe *Actor* car leur sémantique est très proche. La différence est que les parties prenantes du futur produit ne sont pas nécessairement ses futurs utilisateurs. Les acteurs sont toujours des utilisateurs du produit.

Les classes *FeatureOccurrence*, *ModelElementInstance* et *Product* du méta-modèle n'ont pas fait l'objet de création de stéréotypes car elles sont instanciées uniquement lors de la configuration d'un produit. Le rôle du profil UML est de créer un modèle de caractéristiques, et non de le configurer.

4.4.2 La navigabilité des classes du modèle métier

Un produit issu d'une ligne de produits logiciels utilise un modèle de données représentant les données qu'il traite, aussi appelé modèle métier, ou modèle de contexte de la ligne de produit. Nous avons choisi de décrire ce modèle à l'aide de diagrammes de classes UML dans RSA car cela constitue un standard industriel. D'autre part, il est possible, facilement, de transformer un modèle UML vers un autre format et inversement.

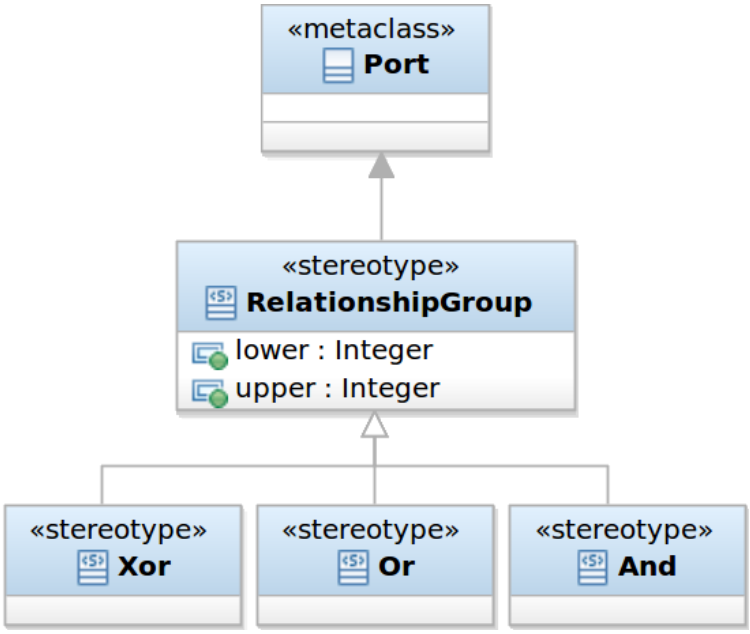

FIGURE 4.16 : Stéréotype *RelationshipGroup*

Comme expliqué dans le chapitre 2, les caractéristiques d'un logiciel sont liées à des données du contexte :

- Certaines instances des classes de ce modèle nécessitent la présence de caractéristiques dédiées à leur traitement. Par exemple, la présence d'une gestion technique centralisée (GTC) dans un bâtiment nécessite que le logiciel possède une caractéristique "interface de communication" compatible avec cette GTC.
- Certaines caractéristiques nécessitent la disponibilité de données particulières. Par exemple, une caractéristique de gestion de la température nécessite la présence de capteurs de température et de présence.
- Certaines caractéristiques dépendent de la présence de données hiérarchisées. Par exemple, une caractéristique relative à une pièce dépend de la sélection d'une caractéristique relative à la CTA qui gère cette pièce. Plus précisément, la caractéristique *Optimisation de la température de l'air soufflé* relative à une pièce ne peut être sélectionnée que si la caractéristique *Contrôle de la température de l'air soufflé* relative à la CTA est sélectionnée. Ce choix doit être fait pour chaque CTA du bâtiment et chaque pièce gérée par l'une de ces CTA. Un modèle métier ne permet pas de déterminer quelles associations décrivent cette hiérarchie parmi toutes les associations du modèle.

Il est donc nécessaire d'enrichir la sémantique du modèle en marquant quelles associations décrivent la hiérarchie des classes associées à des caractéristiques. Nous proposons un profil UML permettant de décrire, pour tout modèle métier modélisé en UML, les relations hiérarchiques, ou, navigabilité entre les classes qui le composent.

Les stéréotypes de ce profil sont indépendants du méta-modèle de caractéristiques car ils concernent uniquement les classes du contexte de la ligne de produits. Ce profil permet

d'enrichir la sémantique d'un modèle métier, sans le modifier, en apportant des informations concernant la navigabilité entre ses classes.

La figure 4.17 présente le méta-modèle relatif à ce profil. Une classe faisant partie de

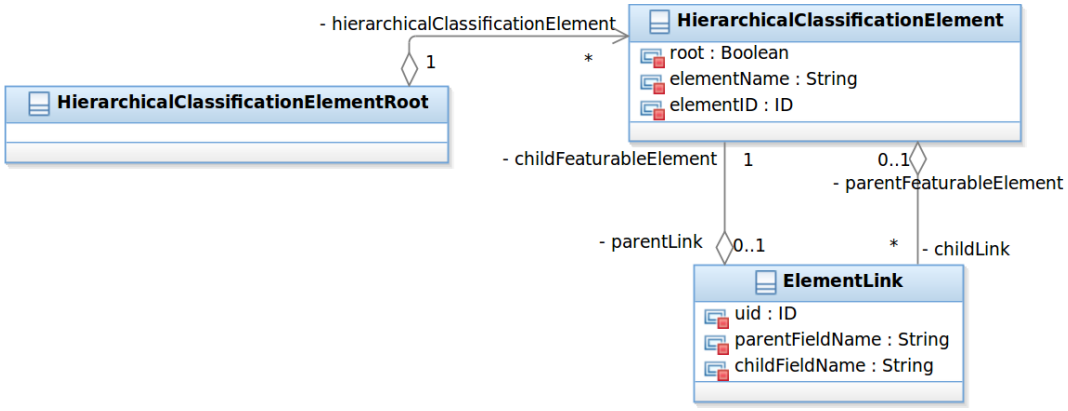

FIGURE 4.17 : Méta-modèle des éléments navigables

la relation hiérarchique est nommée *HierarchicalClassificationElement*. Une association représentant un lien entre deux classes de classification hiérarchique est nommée *ElementLink*.

Le profil UML correspondant contient deux stéréotypes, présentés dans la figure 4.18. Le stéréotype *HierarchicalClassificationElement* s'applique aux classes utilisées pour dé-

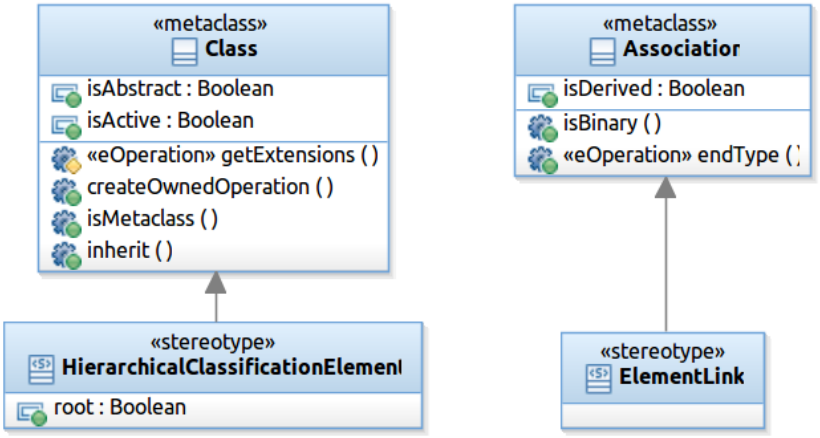

FIGURE 4.18 : Profil UML décrivant la navigabilité entre les classes du contexte

crire des classes du contexte pouvant être associés à des caractéristiques. Le stéréotype *ElementLink* s'applique aux associations entre classes de classification hiérarchique. Le sens de la hiérarchie est donné par l'attribut *aggregation :AggregationKind* et l'opération *isNavigable() :Boolean* des propriétés possédées par l'association. Une association de la hiérarchie nécessite que l'extrémité du côté de la classe fille de la hiérarchie soit navigable et que l'extrémité côté classe parent ait un type d'agrégation différent de *none* (les autres littéraux de l'énumération *AggregationKind* étant *shared* et *composite*).

4.4.3 Outillage

Nous avons choisi de baser l'implémentation de l'outil permettant de créer nos modèles de caractéristiques sur RSA. Cela nous permet de bénéficier d'un environnement de modélisation UML complet et extensible. Les outils de modélisation de RSA se basent sur le framework de modélisation d'Eclipse (Eclipse Modelling Framework – EMF), et fournissent un ensemble d'outils et d'assistants permettant de faciliter la création d'extensions (nouveaux plug-ins). Nous avons donc développé un plug-in permettant la création de modèles de caractéristiques dans RSA.

Cette approche nous permet de définir, avec le même outil, à la fois le modèle métier et le modèle de caractéristiques. Cela permet de bénéficier d'un espace de travail homogène dans lequel il est aisé de décrire les liens entre caractéristiques et classes métier.

La création d'un plug-in intégrant un profil UML dans RSA permet notamment de spécifier quels stéréotypes apparaîtront dans les barres d'outils et palettes de RSA, et de paramétrer la façon dont ils seront présentés dans les diagrammes. La création d'un tel outil consiste à :

1. créer un profil UML (décrit dans la section précédente),
2. générer les modèles décrivant l'outil (les palettes, barres d'outils, ...),
3. générer le code du plug-in, l'éditer si nécessaire,
4. compiler et déployer le plug-in.

Dans notre cas, nous avons besoin uniquement d'un éditeur de modèles de caractéristiques. Nous nous sommes concentrés sur le développement de la "palette" de cet éditeur. Il s'agit d'une barre de menus permettant un accès direct à des éléments spécifiques à un type de diagramme ou un type d'éditeur.

La figure 4.19 montre l'un des diagrammes décrivant la structure de la palette. Chaque élément du diagramme est une classe stéréotypée générée à partir des stéréotypes du profil UML. Nous avons construit plusieurs diagrammes, similaires à celui présenté dans la figure 4.19, décrivant la structure de tous les menus de la palette du plug-in. La catégorie nommée *Feature hierarchy* contient quatre sous-catégories et l'élément *Feature*. Chacune des sous-catégories contient des éléments utilisables dans des diagrammes. Par exemple, la sous-catégorie *Hierarchy* contient les éléments *Implement*, *Enrich* et *Detail*. Ces éléments de la palette permettront ensuite de créer les associations avec les stéréotypes correspondants.

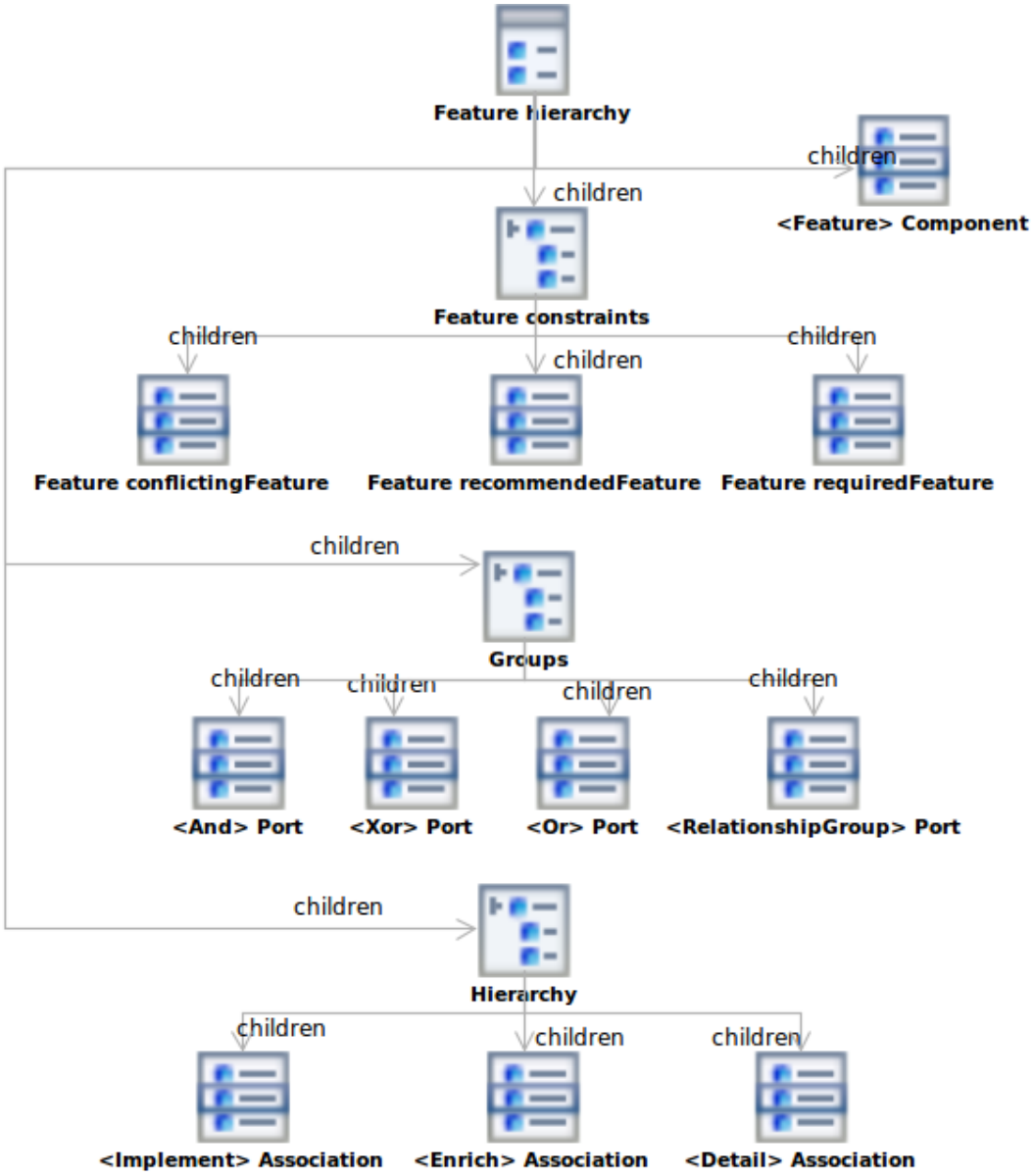

FIGURE 4.19 : Organisation des éléments de la palette concernant l'organisation hiérarchique des caractéristiques

Le résultat final, après avoir généré le code et déployé le plug-in, est présenté dans la figure 4.20. Elle montre la catégorie *Feature hierarchy* "dépliée" dans la palette du plug-in. Ainsi, tous les stéréotypes nécessaires à la création de modèles de caractéristiques sont directement accessibles. Chacun des éléments de la palette donne accès à l'un des stéréotypes du profil ou à l'une des tagged values d'un stéréotype. Par exemple, l'élément de la palette *Feature requiredFeature* permet d'associer une caractéristique à une caractéristique qu'elle requiert. Une référence vers la caractéristique requise est ajoutée dans la tagged value de la caractéristique source de la relation.

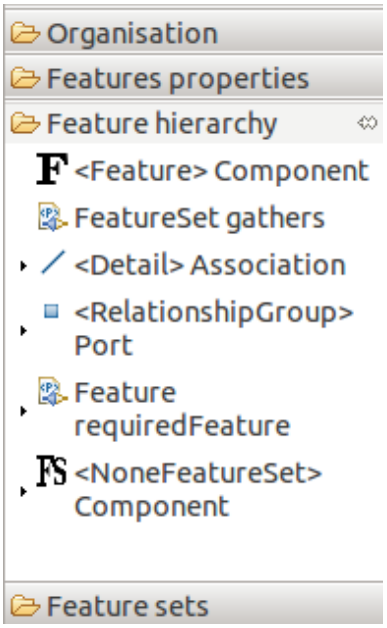

FIGURE 4.20 : Vue de la palette permettant d'éditer des modèles de caractéristiques

La figure 4.21 présente l'ensemble des éléments présents dans la palette de notre plug-in. Nous y retrouvons l'ensemble des stéréotypes décrits précédemment plus des éléments permettant d'affecter des valeurs aux attributs des stéréotypes. Par exemple, l'élément *Feature set gathers* renvoie à l'attribut *gatheredFeatures* du profil et du méta-modèle (montré dans la figure 4.10). La colonne de gauche représente les catégories d'éléments de la palette. Nous avons créé quatre catégories regroupant tous les éléments de la palette :

- *Organisation* : contient les éléments permettant d'organiser la ligne de produits à l'aide d'ensembles de caractéristiques, niveaux, de répartir les caractéristiques entre les parties prenantes, et de décrire les assets associés aux caractéristiques,
- *Feature properties* : contient les éléments permettant d'ajouter des informations aux caractéristiques telles que les propriétés et les liens avec un modèle de contexte,
- *Hierarchy* : contient les éléments permettant de créer la hiérarchie de caractéristiques du modèle. Cette catégorie contient donc l'élément *caractéristique*, les relations hiérarchiques entre caractéristiques, les groupes de relations, et les contraintes entre caractéristiques,
- *Feature sets* : contient les éléments permettant de créer des ensembles de caractéristiques.

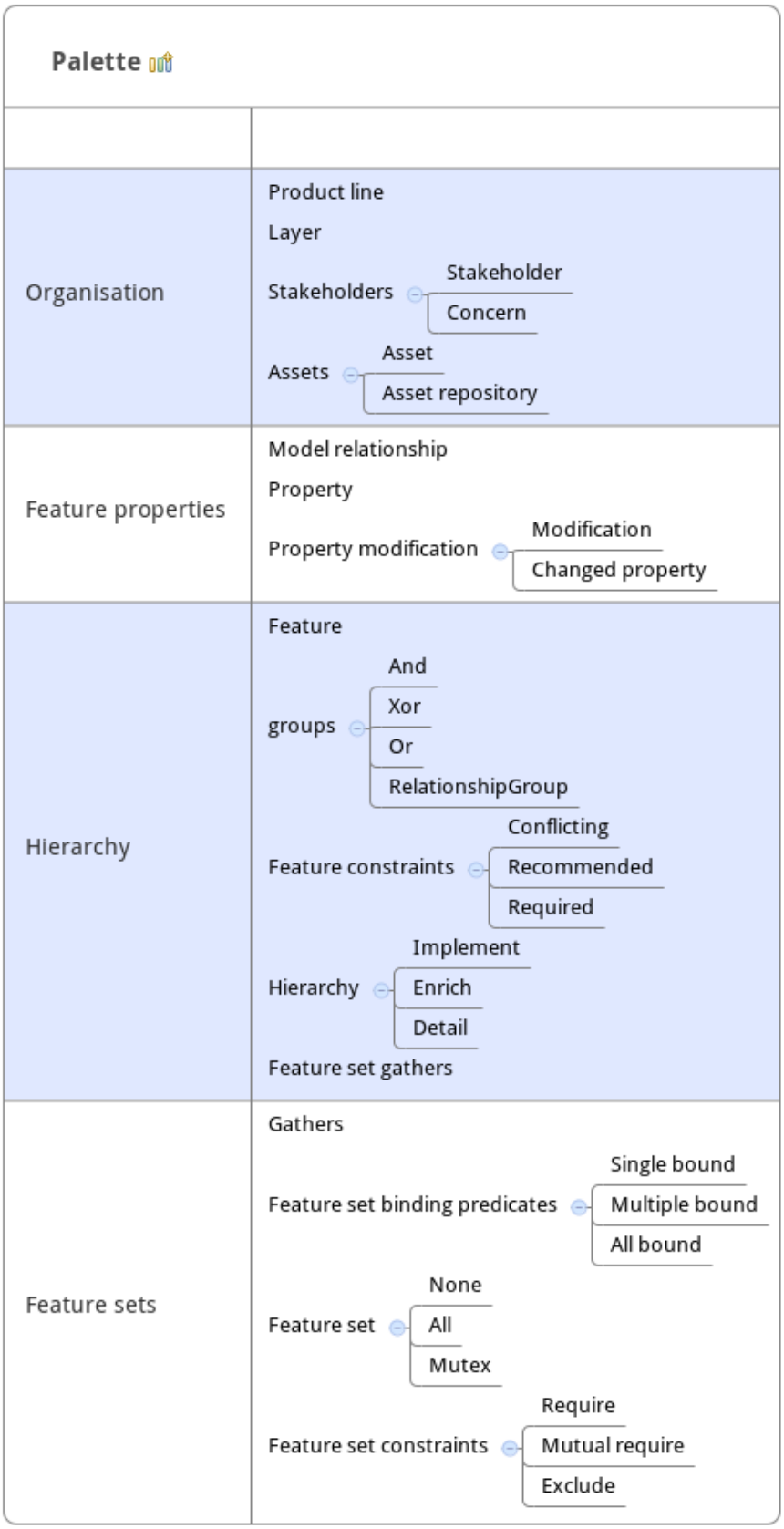

FIGURE 4.21 : Organisation de l'ensemble des éléments de la palette

4.5 Comparaison avec les outils existants

Différentes contributions ont fourni des outils permettant la création de modèles de caractéristiques [University~of~Waterloo; Pros~Labs; T. Bednasch *et al.*, 2003; Acher *et al.*, 2011b]. Cependant elles reposent toutes sur le développement d'un outil dédié permettant la création des modèles de caractéristiques. Nous avons au contraire choisi de réutiliser un outil de modélisation UML dont nous avons étendu les fonctionnalités au moyen d'un profil UML. L'avantage de notre approche est que nous avons pu créer l'éditeur assez rapidement pratiquement sans programmation. Nous avons développé un prototype, basé sur les algorithmes décrits dans le chapitre 5, permettant d'adapter le modèle de caractéristiques à un contexte et de le configurer. Ce prototype pourrait être intégré au plugin RSA contenant le profil UML dans le cadre d'un développement futur.

Plusieurs travaux [Gomaa et Shin, 2007; Hassan Gomaa et Michael E. Shin, 2010] montrent que les approches basées sur UML, et intégrées dans un logiciel de modélisation sont viables. Cela permet par exemple de mapper les classes du logiciel avec des caractéristiques afin de générer du code correspondant aux caractéristiques sélectionnées. Les travaux de Clauß [Matthias Clauß, 2001] montrent également que l'utilisation d'un profil UML pour représenter un modèle de caractéristiques est un moyen efficace de représenter la variabilité d'un logiciel. Mais contrairement à lui, nous avons choisi la méta-classe UML *Component* car elle possède une sémantique plus riche et représente, comme une caractéristique, un élément constitutif d'un système et fournissant un service.

4.6 Conclusion

Notre méta-modèle de caractéristiques est développé pour convenir aux besoins que nous avons identifiés dans un projet industriel. Nous y avons inclus des nouveautés permettant de décrire les liens entre caractéristiques et classes métier, et les concepts de *layers*, *stakeholder concerns*, *feature-sets*, et *group constraints*.

Nous avons créé un outil en étendant les fonctionnalités de RSA permettant de produire des modèles de caractéristiques liés au modèle métier utilisé dans le projet. Nous avons également créé des transformations de modèles permettant d'extraire de RSA les modèles de caractéristiques au format XML. Cette approche nous a permis de nous affranchir de RSA pour développer un prototype faisant la transformation de modèles (présentée dans le chapitre 5).

Une perspective de développement consisterait à réaliser dans RSA la génération du modèle de caractéristiques spécifiques au contexte en créant un plugin s'appuyant sur la représentation interne des modèles UML dans RSA avec Eclipse Modelling Framework (EMF).

Nous pourrions pousser l'intégration de notre profil dans RSA encore plus loin, en utilisant le gestionnaire de dépôts d'assets de RSA pour identifier les assets et les artefacts pouvant être associés aux caractéristiques. Il serait aussi possible de mettre en œuvre une ap-

proche de construction d'un modèle de caractéristiques par rétro-ingénierie des assets et artefact présents dans un dépôt d'assets. Cette approche serait d'autant plus pertinente que la norme RAS [OMG, 2005] (*Reusable Asset Specification*, spécifiant comment décrire et organiser des assets dans un dépôt dédié, dispose d'un mécanisme de gestion de la variabilité entre les artefacts.

D'autre part, une prochaine étape souhaitable consisterait à intégrer notre méta-modèle au *Common Variability Language* (CVL) actuellement en cours de normalisation par l'OMG. Nous pourrions réutiliser leur description de la variabilité et leurs méthodologies pour intégrer la variabilité dans le développement d'un logiciel, et ajouter nos nouveaux concepts permettant d'adapter le logiciel à un contexte.

CHAPITRE

5 Génération automatisée de modèles de caractéristiques dépendants du contexte par transformation de modèles

Sommaire

5.1 Introduction **67**
5.2 Description globale de l'approche **68**
5.3 Définitions et notations **69**
5.4 Algorithme de génération du modèle de caractéristiques spécifique au contexte **88**
5.5 Implémentation **92**
5.6 Conclusion **93**

5.1 Introduction

Ce chapitre explique notre méthode d'adaptation de modèles de caractéristiques à un contexte donné. Il s'agit d'une méthode générique s'appuyant sur une démarche d'ingénierie dirigée par les modèles. Elle automatise l'adaptation d'un modèle de caractéristiques en fonction d'un modèle de contexte décrivant l'environnement d'exécution ou de déploiement du nouveau produit. Plus globalement, il s'agit du contexte selon lequel des choix de configuration vont être faits.

La section 5.2 présente globalement les différents modèles que nous utilisons ainsi que le fonctionnement de notre méthode. La section 5.3 définit l'ensemble des concepts et une notation que nous utilisons dans la suite de ce chapitre. La section 5.4 présente l'algorithme permettant la génération d'un modèle de caractéristiques adapté à un contexte. La section 5.5 décrit l'implémentation de notre méthode. La section 5.6 conclut ce chapitre.

5.2 Description globale de l'approche

Notre approche intègre les cinq modèles présentés dans la figure 5.1. Le *context model* (CM), ou modèle du contexte, décrit les concepts du contexte que le produit utilise. Ce modèle peut être créé à l'aide d'un langage spécifique à un domaine (DSL) ou d'un modèle UML. Dans le cadre de notre étude de cas, ce modèle est le modèle d'infrastructure de bâtiments, qui répertorie l'ensemble des concepts du domaine du bâtiment, par exemple, capteur de température, salle de réunion, ou salle de cours.

Le *Generic Feature Model* (FM), ou modèle générique de caractéristiques, décrit les caractéristiques ainsi que les contraintes à valider pour autoriser leur présence dans un produit. Notre modèle générique de caractéristiques permet l'association de caractéristiques génériques (GF) à des concepts du contexte afin de spécifier que certaines d'entre elles doivent être dupliquées selon les instances de ces concepts du contexte. Dans notre approche, ce modèle n'est pas utilisé directement pour configurer un nouveau produit. Il doit d'abord être adapté selon un contexte spécifique, décrit par le modèle d'instances du modèle de contexte, ou *Context Model Instance*. Dans notre étude de cas, le FM décrit l'ensemble des caractéristiques du logiciel d'optimisation énergétique et en associe certaines à des concepts du contexte du bâtiment.

Le *Context Model Instance* (CMI), ou instance du modèle de contexte, décrit les éléments du contexte qui sera géré spécifiquement par un nouveau produit donné. Dans notre étude de cas, il s'agit de la représentation du bâtiment qui sera géré par un nouveau logiciel d'optimisation énergétique. Une représentation de bâtiment contient par exemple des éléments qui décrivent les salles du bâtiment avec notamment leurs dimensions et les capteurs installés.

Le *Context Specific Feature Model* (CSFM), ou modèle de caractéristiques spécifique au contexte, contient l'ensemble des caractéristiques spécifiques au contexte, ou *Context Specific Feature* (CSF), pouvant être choisies pour construire un nouveau produit dans un contexte d'exécution donné. Ce modèle est spécifique à un FM et à un CMI car chaque CSF fait référence à une GF et, s'il y a lieu, à une instance de concept du contexte. Ce modèle sert à décrire quelles sont les caractéristiques que le logiciel peut avoir pour traiter les éléments d'un contexte donné. associées. Par exemple, dans notre étude de cas, des CSF existent pour chaque caractéristique du logiciel applicable à chaque salle du bâtiment. Les caractéristiques présentes dans ce modèle doivent valider les contraintes liées à l'instance du contexte auxquelles elles sont Dans ce chapitre, nous développons une approche originale pour générer automatiquement ce modèle.

Le *Product Configuration*, ou configuration de produit, représente un ensemble de CSF ayant été choisies par les commanditaires pour composer un futur produit. Pour être valide, une configuration doit satisfaire toutes les contraintes de variabilité du modèle de caractéristiques.

La figure 5.2 synthétise l'utilisation des modèles de notre approche. Les modèles CM,

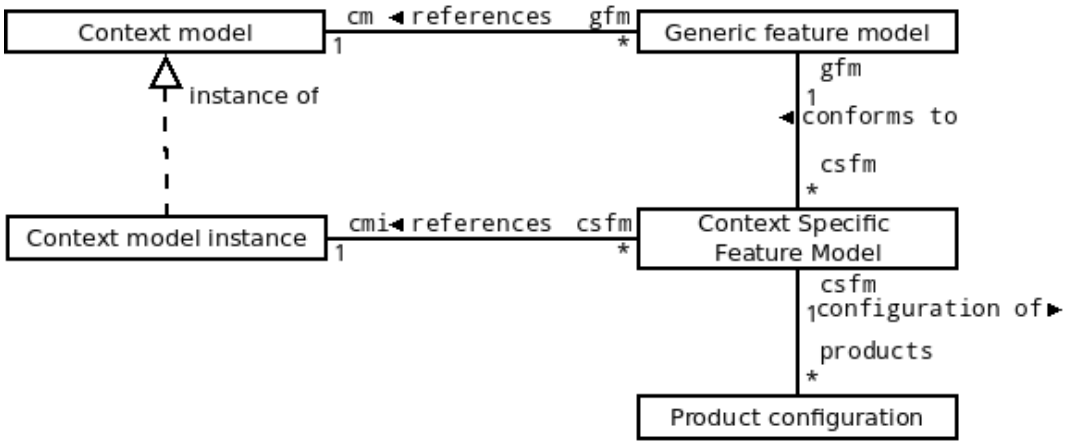

FIGURE 5.1 : Vue d'ensemble des modèles de notre approche

FM, et CMI sont fournis à notre processus en entrée. Le processus crée un CSFM en sortie qui peut ensuite être utilisé par un utilisateur pour créer une configuration de produits.

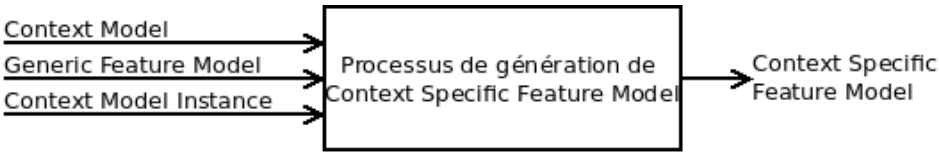

FIGURE 5.2 : Utilisation des différents modèles dans notre processus de génération de modèle de caractéristiques

Dans la section suivante, nous définissons chacun des modèles mentionnés ci-dessus.

5.3 Définitions et notations

Cette section décrit à l'aide d'un formalisme mathématique les modèles utilisés dans notre approche. Dans un premier temps, nous décrirons les contextes dans lesquels s'exécutent les produits de la ligne de produits. Dans ces contextes, des éléments sont associés à d'autres éléments. La multiplicité, inspirée des langages tels que Entité-Association ou UML, servira à décrire, pour une relation *r* donnée et pour un élément *e* donné, à combien d'éléments l'élément *e* peut être associé par *r*. La multiplicité est décrite par un intervalle :

- sa borne inférieure est le nombre minimum d'éléments auxquels *e* doit être associé par *r*,
- sa borne supérieure est le nombre maximum d'éléments auxquels *e* peut être associé par *r*.

Cette notion de multiplicité apparaîtra aussi dans les modèles de caractéristiques. Elle signifiera que la caractéristique peut être dupliquée un nombre de fois inclus dans l'intervalle défini par sa multiplicité.

Nous définissons de façon préliminaire la notion de multiplicité. Nous réutiliserons cette notation dans les définitions suivantes.

Définition 1. Multiplicité. *L'ensemble des multiplicités est* $\mathrm{Mult} = \mathbb{N} \times \mathbb{N} \cup \{*\}$. *De plus, nous définissons :*

- Min_M : *Mult* $\rightarrow \mathbb{N}$ *l'application qui, à une multiplicité, associe sa borne inférieure. Pour* $(m_n, m_x) \in$ *Mult*, $\mathrm{Min}_M((m_n, m_x)) = m_n$,
- Max_M : *Mult* $\rightarrow \mathbb{N} \cup \{*\}$ *l'application qui, à une multiplicité, associe sa borne supérieure. Pour* $(m_n, m_x) \in$ *Mult*, $\mathrm{Max}_M((m_n, m_x)) = m_x$.

Nous formalisons dans la notion de modèle de contexte la description des concepts et des relations que nous trouvons dans un domaine d'application sur lequel portera la ligne de produits.

Un modèle de contexte capture l'essentiel de ce qui nous intéresse et reprend l'esprit des langages de modélisation tels que Entité-Association, UML, ou ontologiques, et nous permet de décrire notre approche d'une manière indépendante de tel ou tel langage de modélisation spécifique.

Afin de donner un cadre général à l'approche, nous avons choisi de le modéliser sous la forme d'un multigraphe orienté et étiqueté. Chaque nœud représente un concept du contexte. Chaque arc représente une relation entre deux concepts.

Définition 2. Modèle de contexte (CM). *Un modèle de contexte décrit les informations sur les concepts utilisées par le futur produit.*
Il s'agit d'un septuplet $CM = (G, V_C, V_A, l_C, l_A, \mathrm{mult}_{CM}, N)$, *avec :*

- $G = (C, A, s, t)$ *est un multigraphe orienté et étiqueté*
  - C *est un ensemble de nœuds.*
  - A *est un ensemble d'arcs.*
  - $s : A \rightarrow C$, *associe à un arc son nœud source.*
  - $t : A \rightarrow C$, *associe à un arc son nœud cible.*
- V_C *et* V_A *définissent des ensembles finis d'étiquettes respectivement pour les nœuds et les arcs.*
- $l_C : C \rightarrow V_C$ *est une fonction qui associe à un nœud son étiquette.*
- $l_A : A \rightarrow V_A$ *est une fonction qui associe à un arc son étiquette.*
- $\mathrm{mult}_{CM} : A \rightarrow$ *Mult, associe à chaque arc une multiplicité.*
- $N \subseteq A$, *est un ensemble d'arcs du multigraphe G représentant la navigabilité*[1] *entre des nœuds de C. Cet ensemble d'arcs induit un graphe sans circuits.* N *possède la propriété suivante :* $\forall (c_1, c_2) \in C \times C, \exists ! a \in N$ *avec* $s(a) = c_1$ *et* $t(a) = c_2$.

De plus, nous introduisons le prédicat P *(pour Path), défini par* xPy, $x, y \in C$, *avec* xPy *vrai ssi il existe un chemin de* x *vers* y *dans le graphe induit par* N.

La figure 5.3 présente un exemple de modèle de contexte. Les noms des nœuds et des arcs sont représentés en gras, les étiquettes en italique. Les nœuds *c1, c2, c3* et *c4* représentent des concepts du contexte reliés entre eux par des arcs étiquetés par un nom et une multiplicité. Le graphe présenté dans la figure 5.3 est décrit comme suit.

1. Cette notion de navigabilité est différente de la notion de navigabilité en UML.

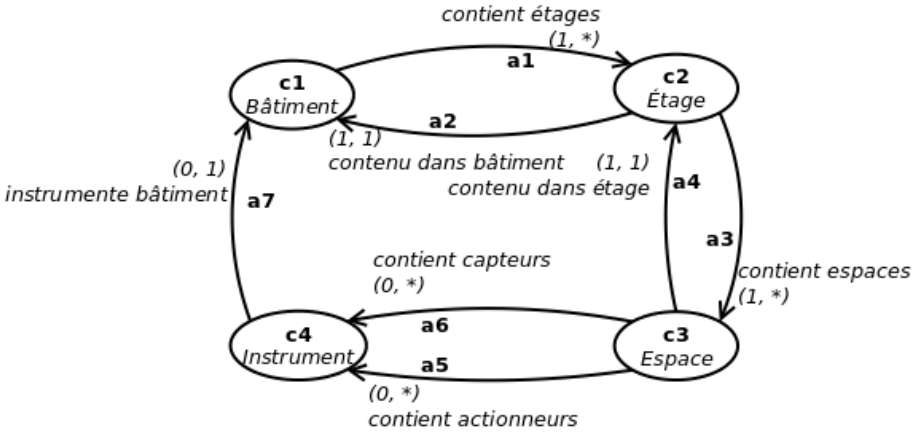

FIGURE 5.3 : Représentation du modèle de contexte

$C = \{c_1, c_2, c_3, c_4\}$
$V_C = \{$*Bâtiment*, *Étage*, *Espace*, *Instrument*$\}$
$V_A = \{$*contient étages*, *contenu dans bâtiment*, *contient espaces*, *contenu dans étage*, *contient capteurs*, *contient actionneurs*, *instrumente bâtiment*$\}$
$l_C(c_1) =$ *Bâtiment*, $l_C(c_2) =$ *Étage*, $l_C(c_3) =$ *Espace*, $l_C(c_4) =$ *Instrument*
$A = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$
$l_A(a_1) =$ *contient étages*, $l_A(a_2) =$ *contenu dans bâtiment*, $l_A(a_3) =$ *contient espaces*, $l_A(a_4) =$ *contenu dans étage*, $l_A(a_5) =$ *contient actionneurs*, $l_A(a_6) =$ *contient capteurs*, $l_A(a_7) =$ *instrumente bâtiment*
$mult_{CM}(a_1) = (1,*)$, $mult_{CM}(a_2) = (1,1)$, $mult_{CM}(a_3) = (1,*)$, $mult_{CM}(a_4) = (1,1)$, $mult_{CM}(a_5) = (0,*)$, $mult_{CM}(a_6) = (0,*)$, $mult_{CM}(a_7) = (0,1)$
$s(a_1) = c_1$, $s(a_2) = c_2$, $s(a_3) = c_2$, $s(a_4) = c_3$, $s(a_5) = c_3$, $s(a_6) = c_3$, $s(a_7) = c_4$
$t(a_1) = c_2$, $t(a_2) = c_1$, $t(a_3) = c_3$, $t(a_4) = c_2$, $t(a_5) = c_4$, $t(a_6) = c_4$, $t(a_7) = c_1$
$N = \{a_1, a_3\}$

Dans cet exemple, on a choisi de naviguer le long des relations d'inclusion entre les lieux (bâtiments, étages, espaces).

Remarquons que, à cause de la propriété sur N, on ne pourrait pas mettre dans N à la fois a_5 et a_6 car ces deux arcs relient les mêmes sommets.

La figure 5.4 présente le graphe sans circuits induit par N. Ainsi, dans cet exemple, le prédicat P retourne les valeurs suivantes $c_1 P c_2 =$ *vrai*, $c_1 P c_3 = vrai$, $c_2 P c_4 = faux$, $c_3 P c_1 = faux$.

Une instance de modèle de contexte est un multigraphe, CMI, défini comme suit. Il décrit des entités d'un niveau d'abstraction inférieur. Là où le modèle de contexte correspond à un modèle UML ou à la TBox d'une ontologie, l'instance de modèle de contexte correspond à un "modèle d'instance" UML ou à la ABox d'une ontologie. Les nœuds de ce graphe sont

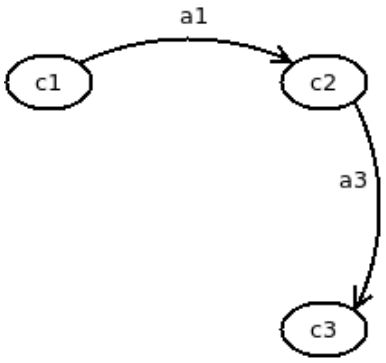

FIGURE 5.4 : Graphe induit par N

des instances de concepts du contexte. Les arcs associent deux instances conformément aux arcs existants entre leurs concepts respectifs comme nous l'expliquerons ci-dessous.

Définition 3. Instance de modèle de contexte (CMI) . *Une instance de modèle de contexte* CMI *décrit des instances et des liens qui se conforment à un modèle de contexte* CM. *Il s'agit d'un quadruplet,* $CMI = (G_I, A_C, A_A, N_I)$*, avec :*

- $G_I = (I, A_I, s_I, t_I)$ *est un multigraphe orienté*
  - *I est un ensemble de nœuds.*
  - A_I *est un ensemble d'arcs.*
  - $s_I : A_I \rightarrow I$*, associe à un arc son nœud source.*
  - $t_I : A_I \rightarrow I$*, associe à un arc son nœud cible.*
- $A_C : I \rightarrow C$ *est une application qui associe à toute instance de I un concept.*
- $A_A : A_I \rightarrow A$ *est une application qui associe à chaque arc entre instances un arc entre concepts auquel il se conforme. Soit un arc* a_1 *entre instances et soit* a_2 *l'arc entre concepts auquel il se conforme, les nœuds source et cible de l'arc* a_1 *doivent être associés respectivement aux nœuds source et cible de l'arc* a_2 *:*

  $$\forall (a_1, a_2) \in A_I \times A \text{ avec } A_A(a_1) = a_2$$

  On a :

  $$\begin{cases} A_C(s_I(a_1)) = s(a_2) \\ A_C(t_I(a_1)) = t(a_2) \end{cases}$$

  Cette application vérifie aussi qu'à chaque arc a *de l'ensemble A, si cet arc est associé à des arcs de* A_I *par* A_A*, alors le nombre de ces arcs est compris dans l'intervalle donné par la multiplicité de l'arc* a*. La figure 5.5 illustre la relation entre la multiplicité* (m_n, m_x) *de l'arc a, et ses instances.*

  $$\forall i \in I, \forall a \in A \text{ avec } mult_{CM}(a) = (m_n, m_x) \text{ et avec } A_C(i) = s(a)$$
  $$m_n \leq |\{a_i \in A_I \text{ avec } s(a_i) = i \text{ et } A_A(a_i) = a\}| \leq m_x$$

- $N_I \subseteq A_I$ *est un ensemble d'arcs qui induit un graphe sans circuits représentant la navigabilité entre les instances de* I*. Il possède la propriété suivante,* $\forall (i_1, i_2) \in I \times I, \exists! a_i \in N_I$ *avec* $s(a_i) = i_1$ *et* $t(a_i) = i_2$*.*

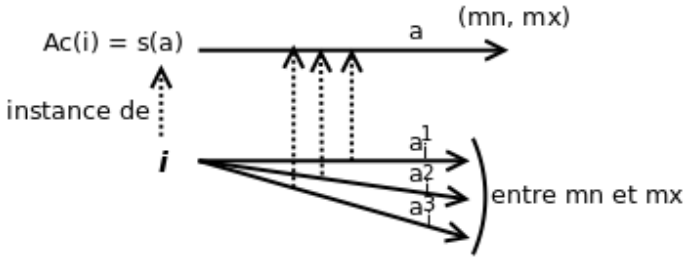

FIGURE 5.5 : Instances d'arcs

- *On note* P_{N_I} *le prédicat tel que* $iP_{N_I}j$, $i,j \in I$, *est vrai ssi il existe un chemin de* i *vers* j *dans le graphe induit par* N_I.
- N_I *est conforme à* N *:* $\forall a_i \in N_I$, *on a* $A_A(a_i) \in N$.

Les figures 5.6 et 5.7 présentent un même modèle instance du modèle de contexte sous la forme d'un graphe. La figure 5.6 présente les étiquettes du graphe, et la figure 5.7 présente le nom des nœuds et des arcs.

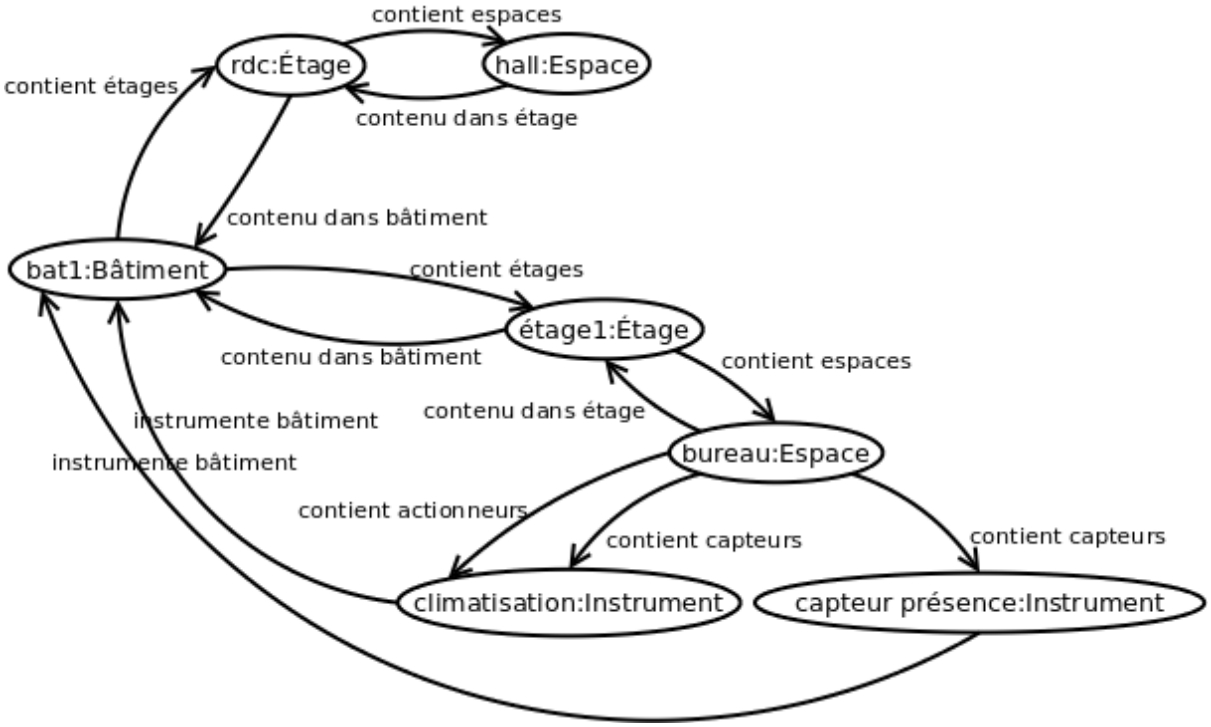

FIGURE 5.6 : Instance du modèle de contexte avec étiquettes

Nous avons nommé les nœuds et les arcs sur le schéma selon les conventions suivantes. D'abord le nom du nœud, suivi de " : ", puis le nom du concept associé selon l'application A_C. Pour des raisons de clarté, les arcs portent ici le nom de l'étiquette de l'arc associé selon l'application A_A.

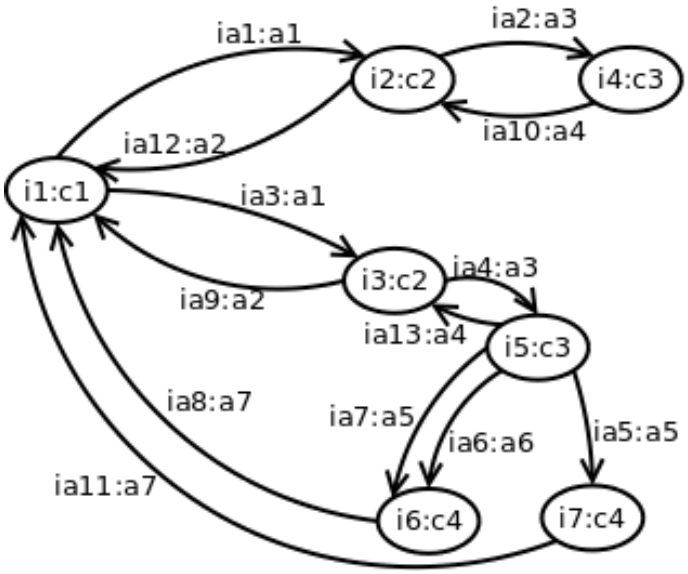

FIGURE 5.7 : Instance du modèle de contexte

Le graphe G est comme suit :

$I = \{i_1, i_2, i_3, i_4, i_5, i_6, i_7\}$
$A_I = \{ia_1, ia_2, ia_3, ia_4, ia_5, ia_6, ia_7, ia_8, ia_9, ia_{10}, ia_{11}, ia_{12}, ia_{13}\}$
$s_I(ia_1) = i_1$, $s_I(ia_2) = i_2$, $s_I(ia_3) = i_1$, $s_I(ia_4) = i_3$, $s_I(ia_5) = i_5$, $s_I(ia_6) = i_5$,
$s_I(ia_7) = i_5$, $s_I(ia_8) = i_6$, $s_I(ia_9) = i_3$, $s_I(ia_{10}) = i_4$, $s_I(ia_{11}) = i_7$, $s_I(ia_{12}) = i_2$,
$s_I(ia_{13}) = i_5$
$t_I(ia_1) = i_2$, $t_I(ia_2) = i_4$, $t_I(ia_3) = i_3$, $t_I(ia_4) = i_5$, $t_I(ia_5) = i_7$, $t_I(ia_6) = i_6$,
$t_I(ia_7) = i_6$, $t_I(ia_8) = i_1$, $t_I(ia_9) = i_1$, $t_I(ia_{10}) = i_2$, $t_I(ia_{11}) = i_1$, $t_I(ia_{12}) = i_1$,
$t_I(ia_{13}) = i_3$

Les concepts associés à chaque instance de I sont comme suit :

$A_C(i_1) = c_1$, $A_C(i_2) = c_2$, $A_C(i_3) = c_2$, $A_C(i_4) = c_3$, $A_C(i_5) = c_3$, $A_C(i_6) = c_4$,
$A_C(i_7) = c_4$

Les arcs entre concepts associés aux arcs entre instances sont les suivants :

$A_A(ia_1) = a_1$, $A_A(ia_2) = a_3$, $A_A(ia_3) = a_1$, $A_A(ia_4) = a_3$, $A_A(ia_5) = a_5$,
$A_A(ia_6) = a_6$, $A_A(ia_7) = a_5$, $A_A(ia_8) = a_7$, $A_A(ia_9) = a_2$, $A_A(ia_{10}) = a_4$
$A_A(ia_{11}) = a_7$, $A_A(ia_{12}) = a_2$, $A_A(ia_{13}) = a_4$
$N_I = \{ia_1, ia_2, ia_3, ia_4\}$

Le graphe sans circuits induit par N_I est présenté à la figure 5.8.

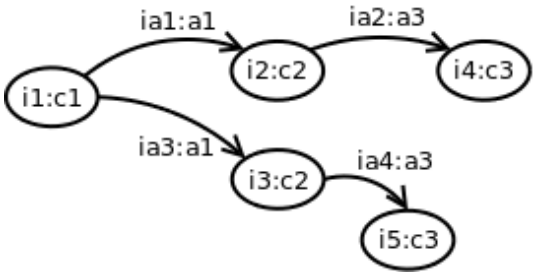

FIGURE 5.8 : Graphe sans circuits induit par N_I

Dans cet exemple, le prédicat P_{N_I} a les valeurs suivantes : $i_1 P_{N_I} i_5 = vrai$, $i_3 P_{N_I} i_4 = faux$.

Nous allons à présent décrire les modèles de caractéristiques. Nous aurons besoin de contraintes que devront respecter les caractéristiques entre elles ou relativement à des éléments d'un contexte. Nous avons choisi de représenter ces contraintes dans des langages inspirés de la logique des propositions et de la logique des prédicats.

Définition 4. Langage de la logique des propositions. *Un langage de logique propositionnelle* $\mathcal{L}_{prop}(Prop)$ *est constitué de symboles contenus dans les ensembles :*

- *Prop, de symboles propositionnels,*
- $L = \{\Rightarrow, \Leftrightarrow, \wedge, \vee, \neg\}$*, de connecteurs logiques.*

L'ensemble des formules propositionnelles $\mathcal{F}$ *est le langage sur l'alphabet* $Prop \cup L$ *défini par les règles suivantes :*

- *si* $p \in Prop$ *alors* p *est une formule propositionnelle (*$p \in \mathcal{F}$*),*
- *si* $a \in \mathcal{F}$ *alors* $\neg a \in \mathcal{F}$*,*
- *si* $a, b \in \mathcal{F}$ *alors* $(a \wedge b) \in \mathcal{F}, (a \vee b) \in \mathcal{F}, (a \Rightarrow b) \in \mathcal{F}$ *et* $(a \Leftrightarrow b) \in \mathcal{F}$*.*

Définition 5. Langage de la logique des prédicats. *Un langage de la logique des prédicats* $\mathcal{L}_{pred}(Cst, Fn, Pn)$ *possède un vocabulaire W constitué de symboles contenus dans les ensembles :*

- V *de variables,*
- Cst *de constantes,*
- $Cn = \{\Rightarrow, \Leftrightarrow, \wedge, \vee, \neg\}$*, de connecteurs,*
- $Q = \{\forall, \exists\}$*, de quantificateurs,*
- Fn *de fonctions,*
- Pn *de prédicats, ou relations.*

Soit $\Sigma = (Cst, Fn, Pn)$ *la signature de ce langage. L'ensemble* T *des termes sur la signature* Σ *est le langage* $\mathcal{A}(\Sigma)$ *tel que :*

- *toute variable* $v \in V$ *est un terme,*
- *toute constante* $c \in Cst$ *est un terme,*
- *si* $t_1, \ldots, t_n$ *sont des termes et* $f \in Fn$ *un symbole de fonction d'arité* n*, alors* $f(t_1, \ldots, t_n)$ *est un terme.*

Une formule atomique est de la forme $p(t_1, \ldots, t_n)$*, où* $p \in Pn$ *est un symbole de prédicat et* $t_1, \ldots, t_n$ *sont des termes.*

L'ensemble des formules $\mathcal{F}$ sur la signature Σ est le langage sur l'alphabet $\mathcal{A}(\Sigma)$ défini par :
- *toute formule atomique est une formule de $\mathcal{F}$,*
- *si $a \in \mathcal{F}$, alors $\neg a \in \mathcal{F}$,*
- *si $a, b \in \mathcal{F}$, alors $(a \wedge b), (a \vee b), (a \Rightarrow b)$ et $(a \Leftrightarrow b) \in \mathcal{F}$.*
- *si $a \in \mathcal{F}$, et $x \in V$ est une variable, alors $\forall x.a \in \mathcal{F}$ et $\exists x.a \in \mathcal{F}$.*

Définition 6. Modèle de caractéristiques (FM). *Un modèle de caractéristiques* FM*, défini pour un modèle de contexte donné* $CM = (G, V_C, V_A, l_C, l_A, mult_{CM}, N)$*, est un 6-uplet* $FM = \langle F, \lambda, r_f, \Psi_f, DE_{fm}, G_{fm}, featureCC, \Psi_c \rangle$*, avec :*

- F *est un ensemble de caractéristiques.*
- $\lambda : F \rightarrow$ *Mult représente la multiplicité d'une caractéristique. Cela représente les nombres minimum et maximum de fois que la caractéristique peut être dupliquée.*
- $r_f \in F$*, est la caractéristique racine.*
- $\Psi_f \subseteq \mathcal{L}_{prop}(F)$ *est un ensemble de contraintes portant sur les caractéristiques, exprimées à l'aide du langage* $\mathcal{L}_{prop}(F)$.
- $DE_{fm} \subset F \times F$ *est l'ensemble des arcs entre les caractéristiques. Munies de* DE_{fm}*, les caractéristiques forment un arbre orienté dont la racine est* r_f*. Lorsque* $(f, f') \in DE_{fm}$*,* f' *est appelée caractéristique fille de* f.
- $G_{fm} = \{(f, (m_n, m_x), S) \ / \ f \in F, (m_n, m_x) \in Mult, S \subset F \text{ et } \forall s \in S, (f, s) \in DE_{fm}\}$ *est l'ensemble des groupes de caractéristiques.*
  La multiplicité d'un groupe de caractéristiques est utilisée pour contraindre la sélection des caractéristiques filles d'une caractéristique parent f*.* S *est un ensemble de caractéristiques filles de* f.
- $featureCC : F \rightarrow \mathcal{P}(C \times Mult)$ *est une fonction associant à chaque caractéristique des couples formés d'un concept du contexte et d'une multiplicité. La multiplicité décrit le nombre de fois que la caractéristique peut être dupliquée pour chaque instance du concept. Pour une caractéristique notée* f *associée à un concept* c_1 *et la multiplicité* (m_n, m_x) *(donc avec* $(c_1, (m_n, m_x)) \in featureCC(f)$*), toutes les caractéristiques filles de* f *doivent être associées au même concept* c_1 *ou*[2] *à un concept* c_2 *accessible depuis* c_1 *selon le graphe induit par N (satisfaisant le prédicat* $c_1 P c_2$*) :*
  $\forall f \in F, \forall c_1 \in C$ *avec* $(c_1, (m_n, m_x)) \in featureCC(f)$ *alors* $\forall f' \in F$ *telle que* $(f, f') \in DE_{fm}$ *alors* $(c_1, (m'_n, m'_x)) \in featureCC(f')$*, ou* $(c_2, (m''_n, m''_x)) \in featureCC(f')$ *avec* $c_2 \in C$ *et* $c_1 P c_2$ *vraie.*

- $\Psi_c : F \times C \rightarrow \mathcal{L}_{pred}(Cst, Fn, Pn)$ *est une fonction associant une contrainte décrite avec le langage* $\mathcal{L}_{pred}$*, à un couple caractéristique/concept, avec*
  - $Cst = C \cup F \cup I \cup V_C \cup V_A \cup N \cup A \cup A_I \cup N_I \cup r_f \cup DE_{fm} \cup G_{fm}$*, est l'ensemble des constantes,*
  - $Fn = \{l_C, l_A, mult_{CM}, s, t, A_C, A_A, s_I, t_I, featureCC, \lambda, \lambda_{G_{fm}}, featureCC_{concept}, featureCC_{mult}\}$*, est l'ensemble des fonctions,*
  - $Pn = \{P, P_{N_I}\}$*, est l'ensemble des prédicats.*

  La contrainte sert à spécifier les critères auxquels doit se conformer l'instance du contexte associée au concept afin de permettre la duplication d'une caractéristique.

2. Notez que l'on peut avoir $\{(c_1, (m_n, m_x)), (c_2, (m'_n, m'_x))\} \subseteq featureCC(f)$

De plus nous étendons la définition précédente avec la notation suivante :

- $\lambda_{G_{fm}} : G_{fm} \rightarrow$ *Mult*, associe à un groupe de caractéristiques sa multiplicité.
- $featureCC_{concept} : C \times Mult \rightarrow C$, associe à un couple concept / multiplicité son concept.
- $featureCC_{mult} : C \times Mult \rightarrow Mult$, associe à un couple concept / multiplicité sa multiplicité.

La figure 5.9 présente différentes situations dans lesquelles sont utilisées les multiplicités. La multiplicité *(1, 2)* entre le concept c_2 et la caractéristique A signifie que pour toute instance de c_2, A doit être présent une ou deux fois. La multiplicité *(1, *)* de A signifie qu'un produit doit contenir la caractéristique A au moins une fois et autant que nécessaire. La multiplicité *(1, *)* de la caractéristique B signifie que B doit être présente au moins une fois si A est choisie. La caractéristique C doit être présente une ou deux fois pour toute instance du concept c_2 comme indiqué par la multiplicité de la fonction *featureCC*.

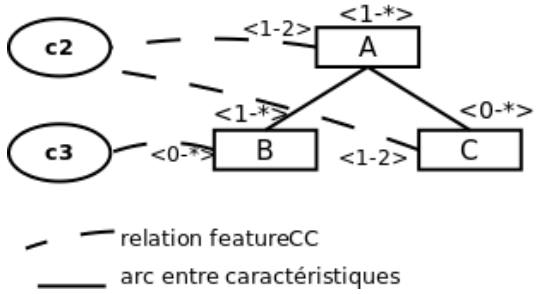

FIGURE 5.9 : Exemple de caractéristiques associées à des concepts du contexte

La figure 5.10 représente un modèle de caractéristiques sous la forme d'un graphe orienté et annoté par les différents éléments de sa définition. Dans cette figure, les arcs entre caractéristiques (DE_{fm}) sont orientés vers le bas. Y sont représentés les caractéristiques, la multiplicité des caractéristiques, les arcs entre caractéristiques de l'ensemble DE_{fm}, les concepts du contexte associés à des caractéristiques, la fonction $featureCC$ associant une caractéristique à un concept et une multiplicité, et l'ensemble des groupes de caractéristiques G_{fm}.

La figure 5.11 représente le modèle de caractéristiques décrit dans la figure 5.10. Nous avons adapté et étendu ici le formalisme plus répandu de Czarnecki *et al.* [2004a]; T. Bednasch *et al.* [2003] et nous allons faire le parallèle avec notre formalisation. Dans ce modèle, la caractéristique B est reliée par un arc avec la multiplicité $(1,2)$ au concept c_2, issu du graphe du modèle de contexte de la figure 5.3. Cela signifie que pour une instance de c_2 on pourra choisir une ou deux B.

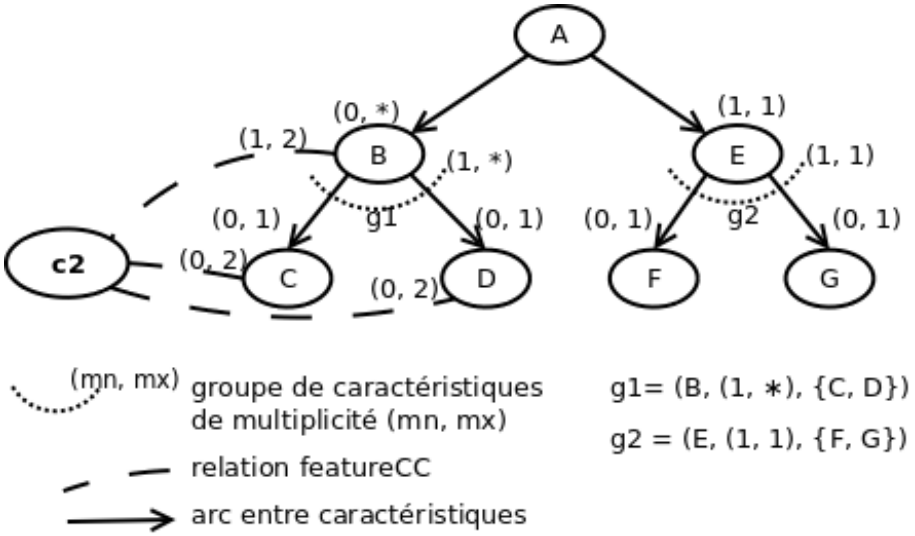

FIGURE 5.10 : Modèle générique de caractéristiques représenté sous la forme d'un graphe orienté

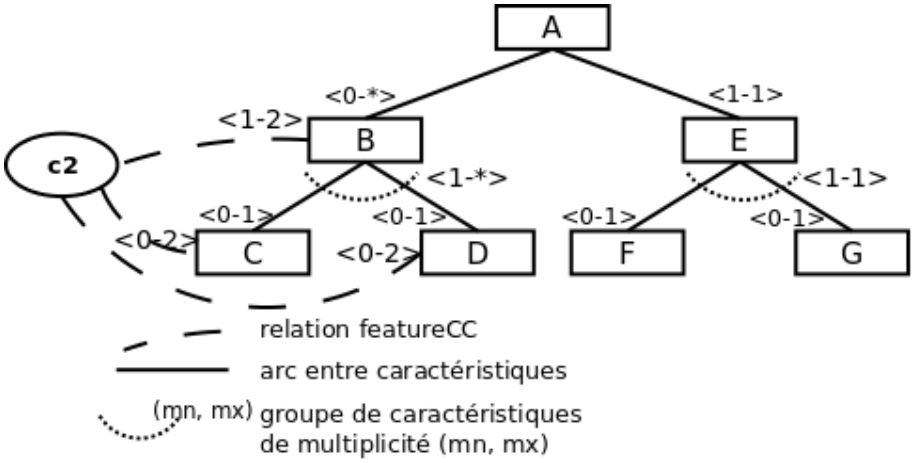

FIGURE 5.11 : Modèle générique de caractéristiques utilisant le formalisme des modèles de caractéristiques avec multiplicité [Czarnecki *et al.*, 2004b] étendu par la liaison avec les concepts du domaine (ici c_2).

Dans le modèle de caractéristiques décrit par les figures 5.10 et 5.11, nous avons les ensembles suivants. L'ensemble des caractéristiques est :

$$F = \{A, B, C, D, E, F, G\}$$

L'ensemble des multiplicités de caractéristiques des figures 5.10 et 5.11 est défini comme suit : A est la racine du modèle de caractéristiques. Sa multiplicité est, par convention, $(1,1)$. La multiplicité des caractéristiques est :

$$\lambda(A) = (1,1)\,;\ \lambda(B) = (0,*)\,;\ \lambda(C) = (0,1)\,;\ \lambda(D) = (0,1)\,;$$
$$\lambda(E) = (1,1)\,;\ \lambda(F) = (0,1)\,;\ \lambda(G) = (0,1)$$

La contrainte suivante exprime que la caractéristique D requiert la caractéristique F :

$$\Psi_f = \{D \Rightarrow F\}$$

Chaque groupe de caractéristiques est défini par la caractéristique parent à laquelle il appartient, une multiplicité spécifiant combien de caractéristiques filles peuvent être choisies, et l'ensemble des caractéristiques groupées. Dans la figure 5.11, un groupe de caractéristiques est décrit par un arc de cercle en pointillés et sa multiplicité. Les caractéristiques C, D sont groupées dans le groupe g_1 sous la caractéristique B avec la multiplicité *(1, *)* et F, G sont groupées dans le groupe g_2 sous la caractéristique E avec la multiplicité *(1, 1)* :

$$g_1 = (B, (1,*), \{C, D\}),\ g_2 = (E, (1,1), \{F, G\})$$
$$G_{fm} = \{g_1,\ g_2\}$$

Les figures 5.10 et 5.11 montrent les caractéristiques B, C et D associées par un arc au nœud du concept c_2. Cet arc correspond à la représentation graphique de la fonction $featureCC$ définie ci-dessus.

$$featureCC(B) = \{(c_2, (1,2))\}$$
$$featureCC(C) = \{(c_2, (0,2))\}$$
$$featureCC(D) = \{(c_2, (0,2))\}$$

Par exemple, dans le cas décrit par la figure 5.12, on a $featureCC(B) = \{(c_2, (1,2))\}$, $featureCC(C) = \{(c_3, (0,*))\}$ et $featureCC(D) = \{(c_2, (1,2))\}$. La relation $featureCC(D) = \{(c_2, (1,2))\}$ est implicite et n'est pas présentée car elle est identique à celle de sa caractéristique parent. Les caractéristiques B et D peuvent être dupliquées 1 ou

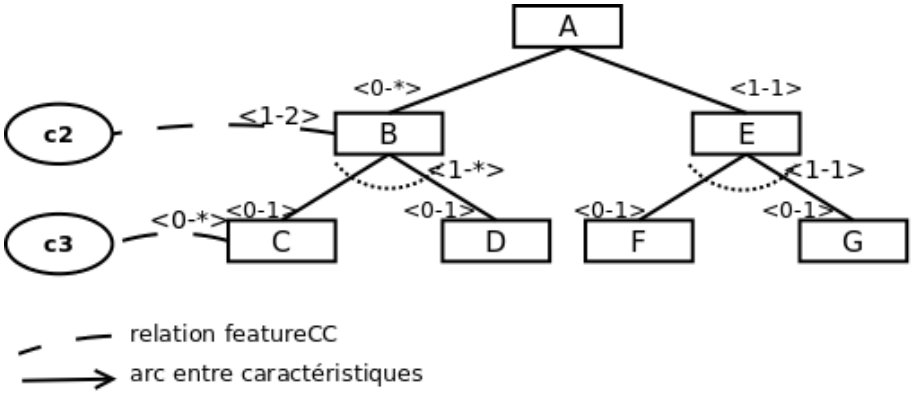

FIGURE 5.12 : Modèle de caractéristiques dans lequel une caractéristique fille est associée à un concept différent de sa caractéristique parent.

2 fois pour chaque instance du concept c_2. La caractéristique C peut être dupliquée 0 ou plusieurs fois pour chaque instance du concept c_3 navigable à partir d'une instance de c_2.

La caractéristique B du modèle de caractéristiques de la figure 5.12 a la multiplicité $(0,*)$. Cette multiplicité spécifie combien de fois B peut être dupliquée dans un produit.

L'association de B au concept c_2, signifie qu'on aura pour chaque instance de c_2 un ou deux exemplaires de la caractéristiques *B*. Le modèle de caractéristiques spécifique à un contexte (CSFM), présenté plus loin, définit le lien entre les exemplaires de caractéristiques et les instances du contexte.

Illustrons à présent Ψ_c. Pour l'exemple, nous avons défini les conditions suivantes pour déterminer quand dupliquer la caractéristique *C* de la figure 5.12. *C* peut être par exemple une procédure d'auto régulation. Il existe une instance $i \in I$ associée à c_3 par la relation A_C. i est le nœud source des arcs x et y. x et y sont associés par la relation A_A respectivement aux arcs a_5 et a_6. La relation A_C doit associer le nœud cible des arcs x et y à un nœud associé à c_4. Ces conditions sont exprimées par la contrainte suivante :

$$\begin{aligned}\psi_c(C,c_3) = \{&\exists i,j \in I, \exists x,y \in A_I \textit{ tels que } A_C(i) = c_3 \textit{ et } A_C(j) = c_4 \textit{ et} \\ &A_A(x) = a_5 \textit{ et } A_A(y) = a_6 \textit{ et } s_I(x) = i \textit{ et } s_I(y) = i \textit{ et } t_I(x) = j \textit{ et} \\ &t_I(y) = j\}\end{aligned} \tag{5.1}$$

La figure 5.13 présente un extrait du graphe du modèle CM et les instances i et j des concepts c_3 et c_4 de ce modèle (l'instance du modèle de contexte est notée CMI). La présence de ces instances de nœuds (i et j) et d'arcs (x et y) permet de satisfaire la contrainte énoncée précédemment.

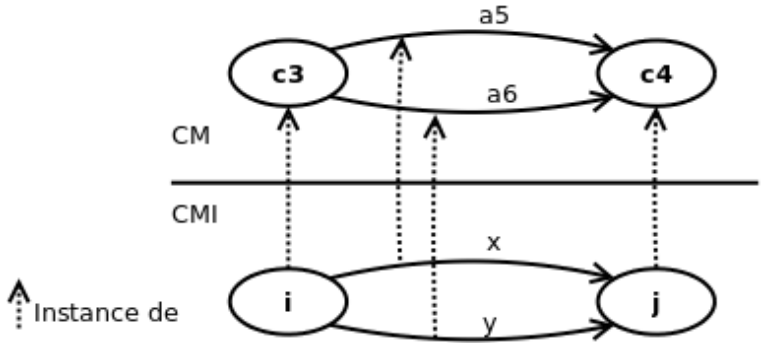

FIGURE 5.13 : Exemple de CMI satisfaisant la contrainte $\psi_c(C,c_3)$

Le CMI présenté dans la figure 5.7 possède deux instances du concept c_3 (i_4 et i_5). Comme le montre la figure 5.14, seule l'instance i_5 permet de valider la contrainte $\psi_c(C,c_3)$ car les arcs a_5 et a_6 sont tous les deux instanciés avec la cible de chacun associée à l'instance i_6 de c_4. i_4 ne valide pas la contrainte car elle n'est associée à aucune instance de c_4.

Pour donner un autre exemple, nous avons défini les conditions suivantes pour déterminer quand dupliquer la caractéristique *C* du diagramme de caractéristiques présenté dans les figures 5.10 et 5.11 : Il existe une instance $i \in I$ associée à c_2 par la relation A_C et une instance $j \in I$ associée à c_3 par la relation A_C. i est le nœud source de l'arc x, et j son nœud cible. j est le nœud source de l'arc y, et i son nœud cible. Les arcs x et y sont associés par la relation A_A respectivement aux arcs a_3 et a_4. Le nœud cible de x doit être le nœud source de y, et inversement, le nœud cible de y doit être le nœud source de x. Ces conditions sont

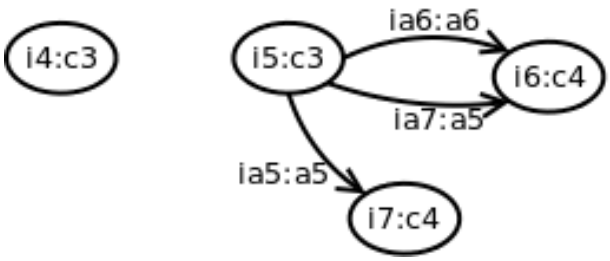

FIGURE 5.14 : Instances du concept c_3 et une partie de leur voisinage (extrait du modèle de la figure 5.7)

exprimées par la contrainte suivante :

$$\psi_c(C,c_2) = \{\exists i,j \in I, \exists x,y \in A_I \textit{ tels que } A_C(i) = c_2 \textit{ et } A_C(j) = c_3 \textit{ et } A_A(x) = a_3 \textit{ et } A_A(y) = a_4 \textit{ et } s_I(x) = i \textit{ et } s_I(y) = j \textit{ et } t_I(x) = j \textit{ et } t_I(y) = i\} \quad (5.2)$$

La figure 5.15 présente un extrait du graphe du modèle CM et les instances i et j des concepts c_2 et c_3 de ce modèle. La présence de ces instances de nœuds (i et j) et d'arcs (x et y) permet de satisfaire la contrainte énoncée précédemment. Le CMI présenté dans la

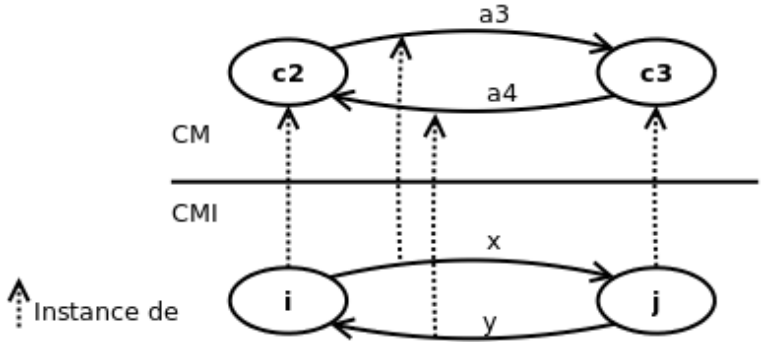

FIGURE 5.15 : Exemple de CMI satisfaisant la contrainte $\psi_c(C,c_2)$

figure 5.7 possède deux instances du concept c_2 (i_2 et i_3). Comme le montre la figure 5.16, les instances i_2 et i_3 permettent de valider la contrainte $\psi_c(C,c_2)$ car les arcs a_3 et a_4 sont tous les deux instanciés avec la cible de l'un correspondant à la source de l'autre.

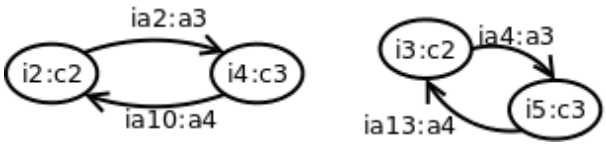

FIGURE 5.16 : Instances du concept c_2 et une partie de leur voisinage (extrait du modèle de la figure 5.7)

La définition suivante formalise le modèle de caractéristiques spécifique à un contexte, ou *Context-Specific Feature Model* (CSFM). Ce modèle contient des caractéristiques spéci-

fiques à un contexte, ou *Context-Specific Features* (CSF), qui représentent des triplets composés d'une caractéristique (GF), d'une instance du contexte et d'une multiplicité.

Définition 7. Caractéristiques spécifiques à un contexte (CSF). *Soient :*

1. *un modèle de contexte* $CM = (G, V_C, V_A, l_C, l_A, \mathtt{mult}_{CM}, N)$,
2. *une instance de ce modèle de contexte,* $CMI = (G_I, A_C, A_A, N_I)$,
3. *un modèle de caractéristiques* $FM = \langle F, \lambda, r_f, \Psi_f, DE_{fm}, G_{fm}, \mathtt{featureCC}, \Psi_c \rangle$.

Une CSF est composée :

- *d'une caractéristique du modèle de caractéristiques,*
- *d'une instance de l'instance de modèle de contexte,*
- *d'une multiplicité.*

L'ensemble des caractéristiques spécifiques à un contexte est $\Phi \subseteq F \times (I \cup \{\varnothing\}) \times \mathtt{Mult}$.

Φ *contient les CSF définies par l'un des deux cas suivants (et contient seulement celles-ci) :*

- *Pour toute caractéristique* f *qui n'est pas associée à un concept du contexte (donc telle que* $\mathtt{featureCC}(f) = \varnothing$*), il existe dans* Φ *une CSF* $\varphi = (f, \varnothing, \lambda(f))$.
- *Pour toute caractéristique* f *associée à un concept* c *du contexte (donc telle que* $(c, (m_n, m_x)) \in \mathtt{featureCC}(f)$*) et pour toute instance* i *avec* $A_C(i) = c$*, il existe dans* Φ *une CSF* $\varphi = (f, i, (m'_n, m'_x))$*, où* $m'_n = \max(m_n, \mathtt{Min}_M(\lambda(f)))$ *et* $m'_x = \min(m_x, \mathtt{Max}_M(\lambda(f)))$*, et* $m'_n \leq m'_x$*. Comme illustré par la figure 5.17, la multiplicité* (m'_n, m'_x) *de la CSF doit être l'intersection*[3] *entre les intervalles des multiplicités de la caractéristique* f *et celle du lien entre* f *et le concept dont* i *est l'instance induit par la fonction* featureCC.

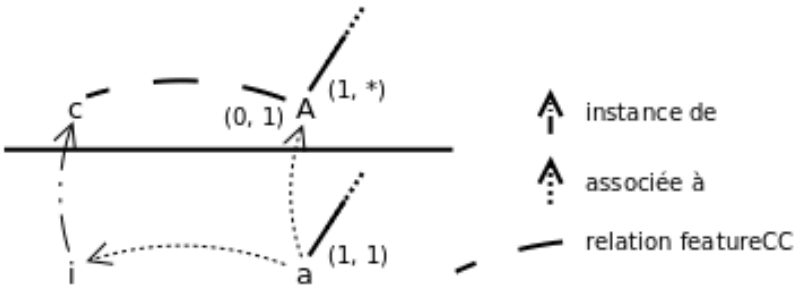

FIGURE 5.17 : Exemple d'intersection entre les intervalles des multiplicités de la caractéristique A et de la fonction featureCC reliant A au concept c dont i est l'instance.

Nous définissons les fonctions auxiliaires suivantes :

- La fonction $\lambda_\Phi : \Phi \to \mathtt{Mult}$ associe sa multiplicité à une CSF. Soit $\varphi = (f, i, (m_n, m_x)) \in \Phi$, $\lambda_\Phi(\varphi) = (m_n, m_x)$.
- $\mathtt{inst} : \Phi \to I \cup \{\varnothing\}$ est une application qui fournit l'instance du contexte associé à une CSF, si elle existe, $\varnothing$ si elle n'existe pas. Pour $\varphi = (f, i, (m_n, m_x)), \varphi \in \Phi, \mathtt{inst}(\varphi) = i$. Pour $\varphi = (f, \varnothing, (m_n, m_x)), \varphi \in \Phi, \mathtt{inst}(\varphi) = \varnothing$.
- $\mathtt{feat} : \Phi \to F$ est une application qui fournit la caractéristique associée à une CSF. Pour $\varphi = (f, i, (m_n, m_x)) \in \Phi$, $\mathtt{feat}(\varphi) = f$.

3. Remarque : si cette intersection est vide, le CSFM ne peut être construit de manière valide.

Nous nommons de deux façons différentes les caractéristiques spécifiques à un contexte :

- une CSF est dupliquée, si elle est associée à une instance du contexte et une caractéristique générique,
- une CSF est non dupliquée, si elle n'est associée à aucune instance, et donc que la caractéristique générique à laquelle elle est associée n'est associée à aucun concept.

Définition 8. Modèle de caractéristiques spécifique à un contexte (CSFM). *Un modèle de caractéristiques spécifique à un contexte, appelé* CSFM[4]*, associé à un CM, un CMI, et un FM donnés avec les notations des définitions 2, 3 et 6, est un quadruplet CSFM* $= \langle \Phi, r_\varphi, DE_{csfm}, G_{csfm} \rangle$*, avec :*

- Φ *est l'ensemble des caractéristiques spécifiques pour* FM, CM *et* CMI *telles que définies à la définition 7.*
- *La caractéristique racine est notée* $r_\varphi \in \Phi$*, et on a* $feat(r_\varphi) = r_f$ et $inst(r_\varphi) = \varnothing$.
- $DE_{csfm} \subset \Phi \times \Phi$ *est l'ensemble des arcs entre CSF.*
  Un arc entre deux CSF est conforme à un arc entre caractéristiques :
  $(\varphi, \varphi') \in DE_{csfm} \Rightarrow (feat(\varphi), feat(\varphi')) \in DE_{fm}$.
  Réciproquement, il existe un arc entre deux CSF s'il existe un arc entre les caractéristiques correspondantes et si la caractéristique de la CSF parent n'est associée à aucun concept ou si les instances auxquelles sont associées les deux CSF sont navigables de l'une à l'autre ou identiques : Soient $\varphi, \varphi' \in \Phi$, $(feat(\varphi), feat(\varphi')) \in DE_{fm}$*, on a* $(featureCC(feat(\varphi)) = \varnothing$ *ou* $(inst(\varphi) P_{N_I} inst(\varphi')$ *ou* $inst(\varphi) = inst(\varphi'))) \Rightarrow (\varphi, \varphi') \in DE_{csfm}$
- G_{csfm} *est un ensemble de groupes de CSF. Un groupe* g_{csfm} *est associé à une CSF parente* φ*, possède une multiplicité dans* Mult *et groupe plusieurs CSF de l'ensemble* $S \subset \Phi$ *tel que* $\forall s \in S$*,* s *est une CSF fille de* φ*. Chaque groupe* g_{csfm} *peut référencer un groupe* g *associé à une caractéristique* f *référencée par* φ.
  Autrement dit l'ensemble des groupes de CSF est $G_{csfm} = \{(\varphi, (m_n, m_x), S, g)$ *tel que* $\varphi \in \Phi$, $(m_n, m_x) \in Mult$, $S \subset \Phi$, $\forall s \in S, (\varphi, s) \in DE_{csfm}$, $g \in G_{fm} \cup \{\varnothing\}$ *et* $f = feat(\varphi)\}$.
  Toutes les CSF associées à une même caractéristique f *et à une même CSF parent, et dont l'instance est associée au même concept sont groupées entre elles. La multiplicité du groupe est égale à la multiplicité de la caractéristique* f *à laquelle elles sont associées. Dans ce cas, il n'y a pas de groupe* g *associé. Soit l'ensemble* S *des CSF telles que* $\varphi \in \Phi$, $feat(\varphi) = f$ *et* $(\varphi', \varphi) \in DE_{csfm}$ *et* $A_C(inst(\varphi)) = c$ *alors* $\exists g_{csfm} \in G_{csfm}$ *avec* $g_{csfm} = (\varphi', \lambda(f), S, \varnothing)$

La figure 5.18 présente un CSFM sous la forme d'un graphe orienté. Chaque CSF est montrée en correspondance avec la caractéristique à laquelle elle est associée. Pour alléger la figure, seul l'arc entre une CSF et l'instance du concept est montré. Les arcs entre les CSF filles et l'instance sont sous-entendus.

Deux CSF associées à la caractéristique *B* ont été créées, pour chaque instance du concept c_2 : *b1* et *b2*. Ces deux CSF ont également des sous-CSF créées à partir des caractéristiques *C* et *D*. *b1* possède les sous-CSF *c1* et *d1* et *b2*, les sous-CSF *c2* et *d2*.

Quatre groupes de CSF ont été créés. Le groupe *g21* fait référence au groupe *g2*.

4. Context-Specific Feature Model

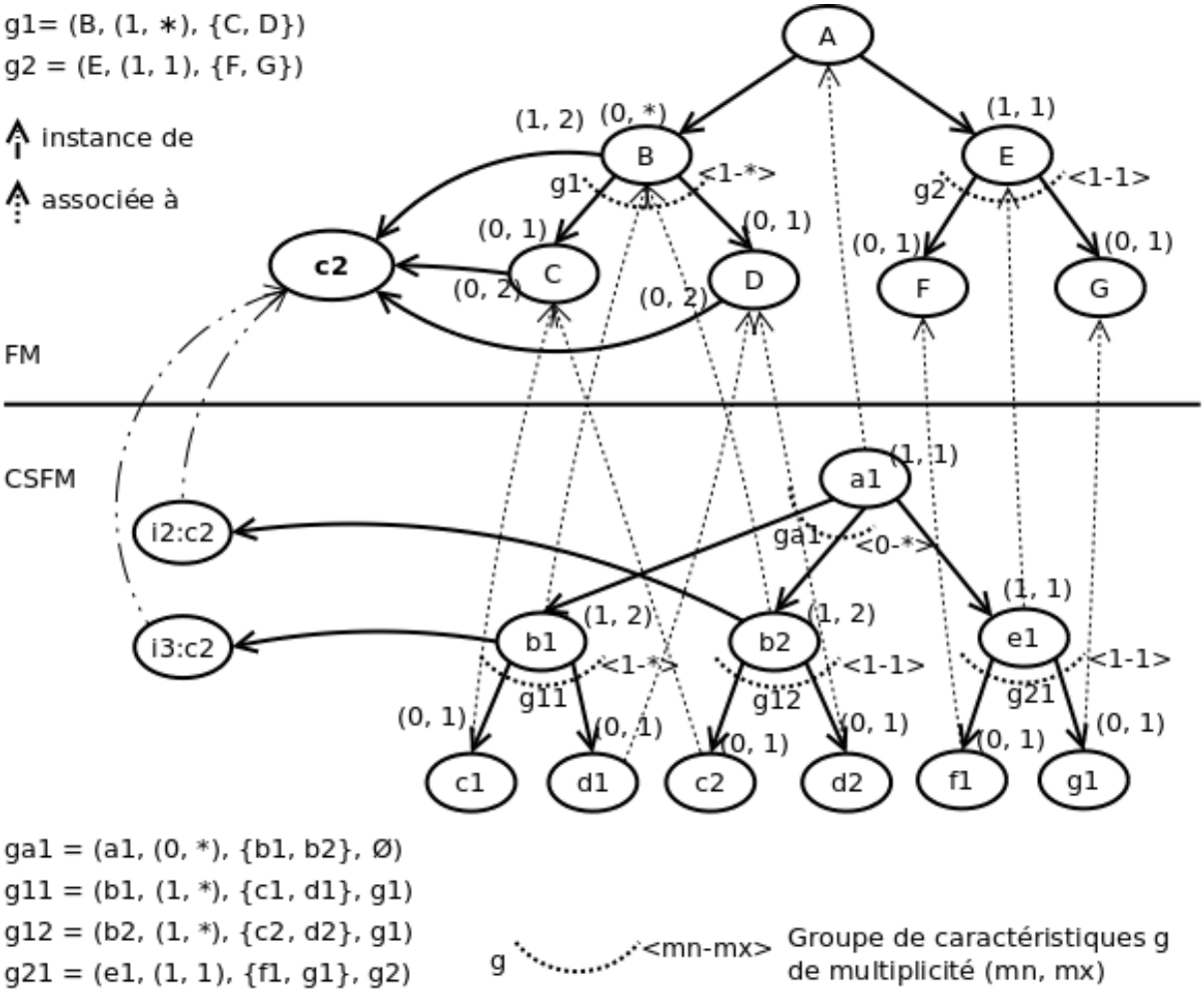

FIGURE 5.18 : Graphe correspondant à un modèle de caractéristiques spécifique au contexte

ga1 regroupe les CSF qui ont été dupliquées à partir de la caractéristique B. La multiplicité de *ga1* est la même que celle de B. Le rôle de ce groupe est d'imposer la sélection d'un nombre de CSF dupliquées conforme à la multiplicité de la caractéristique B.

Les groupes *g11* et *g12* font référence au groupe *g1*. Ils groupent les CSF dupliquées faisant référence respectivement aux instances i_2 et i_3. Leur multiplicité est donc identique à celle de *g1*.

Les multiplicités des CSF dupliquées correspondent à l'intersection des intervalles de la multiplicité de la caractéristique de référence, et de la multiplicité de l'arc entre la caractéristique et son concept associé. C'est pourquoi les CSF *b1* et *b2* ont la multiplicité $(1,2)$, intersection des multiplicités $(1,2)$ et $(0,*)$. De même, les CSF c_1, d_1, c_2 et d_2 ont pour multiplicité $(0,1)$ l'intersection des multiplicités $(0,2)$ et $(0,1)$.

La figure 5.19 décrit le CSFM correspondant à cet exemple. Il s'agit de la représentation du graphe du bas de la figure 5.18 sous la forme, plus conventionnelle, d'un modèle de caractéristiques. Nous avons nommé les CSF dans la figure selon la convention suivante : le

nom de la CSF sous la forme d'une lettre et d'un chiffre, suivi de "-", puis du nom de la caractéristique associée selon la fonction *feat*. Les associations entre les caractéristiques et les objets du contexte ne sont montrées que pour la CSF de niveau hiérarchique le plus élevé. Les associations des CSF filles sont implicites.

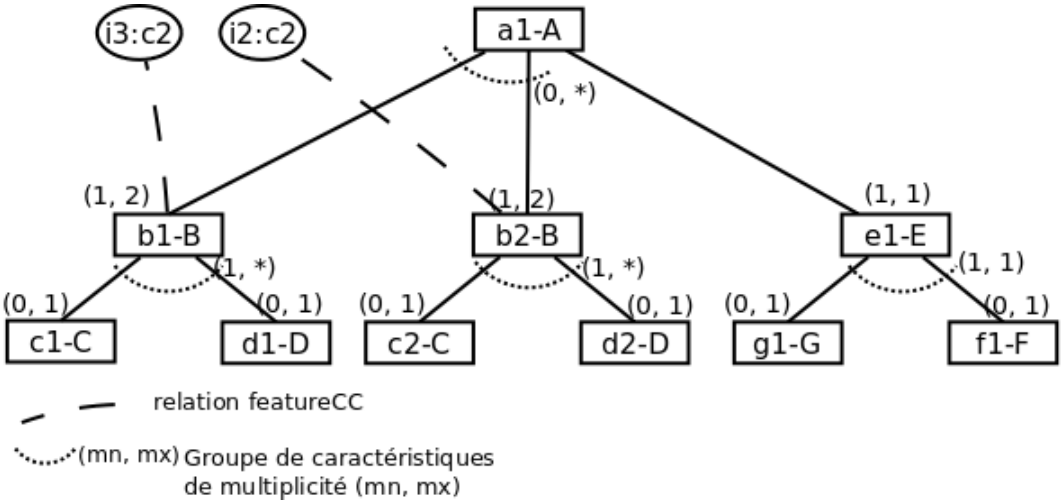

FIGURE 5.19 : Exemple de modèle de caractéristiques spécifique au contexte

Les CSF correspondant au CSFM décrit dans la figure 5.19 sont comme suit :

$$a_1 = (A, \varnothing, (1,1)); e_1 = (E, \varnothing, (1,1)); f_1 = (F, \varnothing, (0,1)); g_1 = (G, \varnothing, (0,1));$$
$$b_1 = (B, i_3, (1,2)); c_1 = (C, i_3, (0,1)); d_1 = (D, i_3, (0,1)); b_2 = (B, i_2, (1,2));$$
$$c_2 = (C, i_2, (0,1)); d_2 = (D, i_2, (0,1))$$

$$\Phi = \{a_1, e_1, g_1, f_1, b_1, c_1, d_1, b_2, c_2, d_2\}$$

Les arcs entre les CSF sont :

$$DE_{csfm} = \{(a_1, e_1), (a_1, b_1), (a_1, b_2), (e_1, g_1), (e_1, f_1),$$
$$(b_1, c_1), (b_1, d_1), (b_2, c_2), (b_2, d_2)\}$$

Les groupes de caractéristiques spécifiques au contexte sont :

$$g_{a_1} = (a_1, (0, *), \{b_1, b_2\}, \varnothing)$$
$$g_{11} = (b_1, (1, *), \{c_1, d_1\}, g_1)$$
$$g_{12} = (b_2, (1, *), \{c_2, d_2\}, g_1)$$
$$g_{21} = (e_1, (1,1), \{f_1, g_1\}, g_2)$$

$$G_{csfm} = \{g_{a_1}, g_{11}, g_{12}, g_{21}\}$$

Nous complétons l'exemple précédent par le diagramme de caractéristiques de la figure 5.20 pour montrer le cas où deux caractéristiques filles d'une même caractéristique parent sont associées à des concepts différents. Nous utilisons les mêmes CM et CMI. Dans le FM, les caractéristiques B et D sont associées au concept c_2 et la caractéristique C, fille de B, est associée à c_3. B peut être présente une ou deux fois pour chaque instance de c_2, et D, zéro ou une fois. B peut être présente zéro ou autant de fois que nécessaire dans tout produit, indépendamment du nombre d'instances de c_2. C peut être présente zéro ou autant de fois que nécessaire pour chaque instance de c_3 et pour chaque CSF associée à B.

Dans le CSFM, nous avons adopté la convention de nommage des CSF consistant à écrire le nom de la caractéristique suivie du nom de l'instance à laquelle elle est associée. Les CSF associées aux caractéristiques A et X ne sont associées à aucune instance. La caractéristique B a donné lieu à la création des CSF $B-i_1$ et $B-i_2$ associées respectivement aux instances i_1 et i_2 du concept c_2. Ces deux CSF, filles de A, sont groupées dans un groupe (inexistant dans le FM) permettant de respecter la multiplicité de la caractéristique B. La multiplicité de ce groupe est donc la même que celle de B : $(0,*)$. La multiplicité des CSF $B-i_1$ et $B-i_2$ a été définie à partir de la multiplicité de la fonction $featureCC$ entre B et c_2, et la multiplicité de B. L'intersection des multiplicités est $(1,2)$. La CSF $B-i_1$ possède trois CSF filles. $D-i_1$, associée à i_1, possède la multiplicité $(0,1)$, intersection de la multiplicité de la relation entre D et c_2 et de la multiplicité de D. Les CSF $C-i_3$, $C-i_4$ et $D-i_1$ sont réunies par un groupe dont la multiplicité est $(1,*)$ et est associé au groupe de même multiplicité dans le FM, ayant en commun les caractéristiques C et D. Les CSF $C-i_3$ et $C-i_4$ sont associées respectivement aux instances i_3 et i_4. Comme elles ont été dupliquées à partir de la caractéristique C, un groupe, dont la multiplicité est identique à celle de C, a été créé pour elles.

Le CSFM décrit l'ensemble des caractéristiques disponibles dans un contexte donné. Les caractéristiques de ce modèle peuvent être choisies par un utilisateur pour configurer un nouveau produit destiné à utiliser les informations du contexte pour lequel le CSFM a été créé. La définition suivante spécifie ce qu'est une configuration valide compte tenu des définitions précédentes.

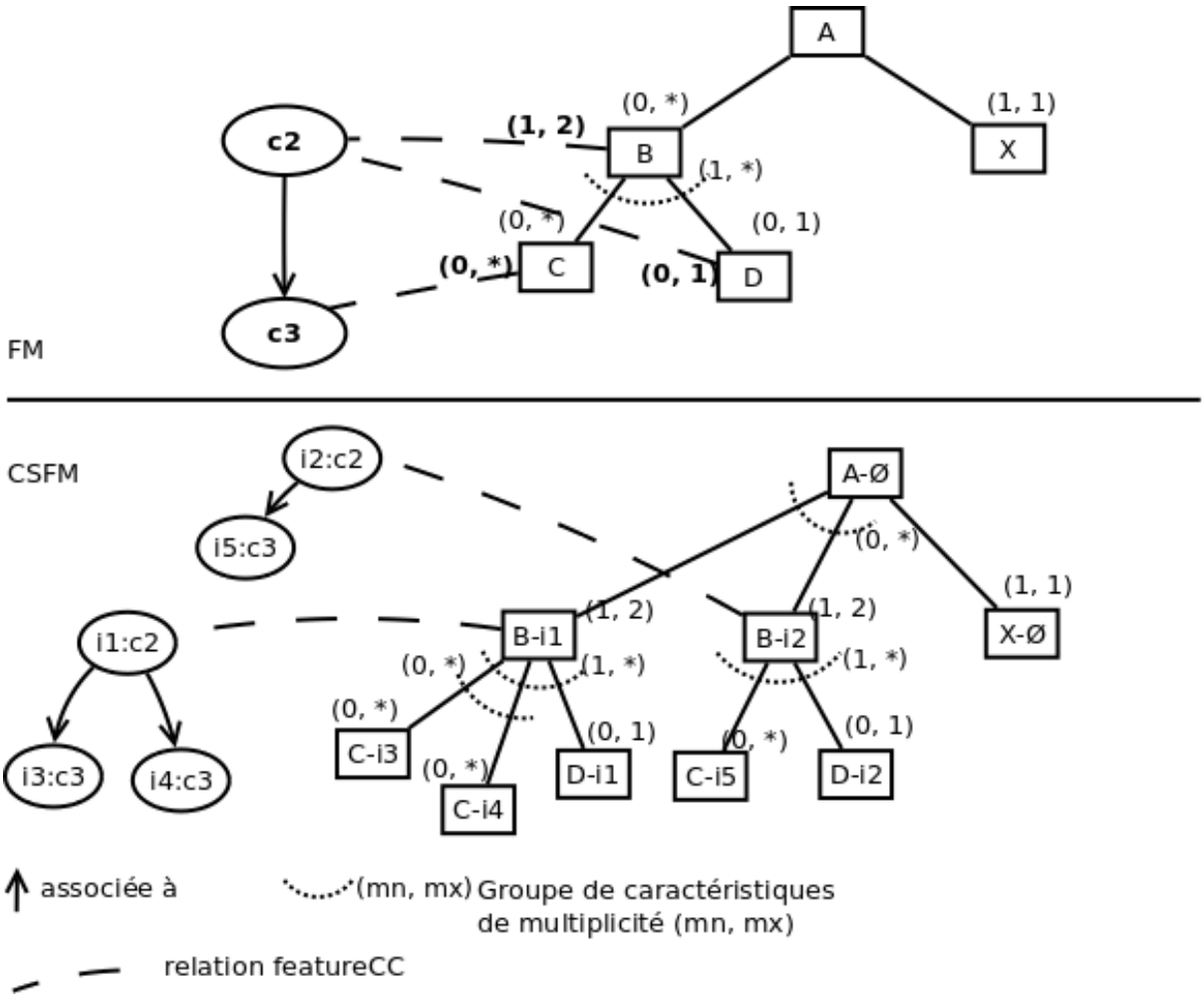

FIGURE 5.20 : Exemple de modèle de caractéristiques spécifique au contexte

Définition 9. Configuration. *Pour un* CM, *un* CMI, *un* FM, *et le CSFM associé* $\langle \Phi, r_\varphi, DE_{csfm}, G_{csfm} \rangle$, *une configuration est un ensemble de CSF de* Φ *choisies par un utilisateur de la ligne de produits. Les CSF choisies doivent satisfaire les contraintes des groupes* G_{csfm}, *de* Ψ_c *et de* Ψ_f *et respecter leurs multiplicités.*

Conformément à ce modèle de caractéristiques spécifiques au contexte, la fonction sémantique [Bontemps *et al.*, 2004] décrivant toutes les configurations valides est notée $[\![CSFM]\!]$. Voici un extrait des configurations valides pour l'exemple de CSFM décrit dans la figure 5.19 :

$$[\![CSFM]\!] = \{\{a_1, e_1, g_1, b_1, d_1, b_2, c_2\}, \{a_1, e_1, f_1, b_1, d_1, b_2, c_2\}, \{a_1, e_1, f_1, b_1, d_1, b_2, d_2\}, \ldots\}$$

Nous pouvons noter que chaque configuration respecte les contraintes énoncées précédemment. Ainsi, il n'est pas possible de choisir la caractéristique c_1 car l'instance du contexte i_3 ne satisfait pas la contrainte $\Psi_c(C, c_2)$ présentée par la formule (5.2). De plus, nous pouvons noter que toutes les configurations doivent inclure les CSF b_1 et b_2 à cause de la multiplicité $(1,2)$ portée sur la relation concept - caractéristique.

La section suivante détaille le fonctionnement de l'algorithme de génération du modèle de caractéristiques spécifique au contexte que nous proposons.

5.4 Algorithme de génération du modèle de caractéristiques spécifique au contexte

L'algorithme de génération du modèle de caractéristiques spécifique au contexte traverse le modèle de caractéristiques par un parcours en profondeur. Pour simplifier la présentation, nous considérons que les modèles *FM, CMI, CM* et *CSFM* (qui est vide à l'initialisation de l'algorithme) sont des variables globales, communes à toutes les fonctions et procédures suivantes.

L'algorithme 1 est la procédure principale de l'algorithme permettant de créer un CSFM à partir des modèles *FM, CMI* et *CM*. Il crée la CSF racine et, pour chaque caractéristique fille de la caractéristique racine de FM, appelle la procédure featureTreeTraversal qui construit par récursivité le reste du CSFM.

Algorithm 1: Procédure principale de l'algorithme de génération du modèle de caractéristiques spécifique au contexte

Input: Les modèles *CM, CMI,* et *FM*
Result: Un modèle *CSFM* construit conformément aux modèles *CM, CMI,* et *FM*

```
// Crée le nœud racine du CSFM.
```
$r_\varphi = (r_f, (1,1), \varnothing)$

```
// Initialise l'ensemble des CSF avec le nœud racine.
```
$\Phi = \{r_\varphi\}$

```
// Initialise l'ensemble des arcs du CSFM.
```
$DE_{csfm} = \{\varnothing\}$

```
// Initialise l'ensemble des groupes de caractéristiques
// du CSFM.
```
$G_{csfm} = \{\varnothing\}$
foreach $f \in F$ *telle que* $(r_f, f) \in DE_{fm}$ **do** `// Pour chaque caractéristique // fille de f`

```
    // Construit récursivement le CSFM.
    FeatureTreeTraversal(f, r_φ)
```

La procédure featureTreeTraversal construit le CSFM de façon récursive par un parcours en profondeur des arcs entre caractéristiques du FM (arcs de l'ensemble DE_{fm}). Deux para-

Procedure featureTreeTraversal($f \in F$, $\varphi_{parent} \in \Phi$)

Input:

$f \in F$, caractéristique à partir de laquelle les caractéristiques du *FM* sont parcourues.

$\varphi_{parent} \in \Phi$, CSF appartenant au modèle de caractéristiques spécifique à un contexte *CSFM*, telle que la caractéristique parent de f est associée à φ_{parent}.

Result:

Actualise le *CSFM* conformément aux modèles *CM*, *CMI*, and *FM*

if $\texttt{featureCC}(f) = \varnothing$ **et** $\texttt{inst}(\varphi_{parent}) = \varnothing$ **then**
- $\varphi = \texttt{addSpecificFeature}(f, \varphi_{parent}, \varnothing)$
- **foreach** $f' \in F$ *telle que* $(f, f') \in DE_{fm}$ **do**
  - $\texttt{FeatureTreeTraversal}(f', \varphi)$

if $\texttt{featureCC}(f) \neq \varnothing$ **et** $\texttt{inst}(\varphi_{parent}) = \varnothing$ **then**
- **foreach** $i \in I$ *avec* $A_C(i) \in \texttt{featureCC}(f)$ **do**
  - $\varphi = \texttt{addSpecificFeature}(f, \varphi_{parent}, i)$
  - **foreach** $f' \in F$ *telle que* $(f, f') \in DE_{fm}$ **do**
    - $\texttt{FeatureTreeTraversal}(f', \varphi)$

if $\texttt{featureCC}(f) \neq \varnothing$ **et** $\texttt{inst}(\varphi_{parent}) \neq \varnothing$ **then**
- **foreach** $i \in I$ *avec*
  $(i = \texttt{inst}(\varphi_{parent})$ **or** $\texttt{inst}(\varphi_{parent}) P_{N_I} i)$ **and** $A_C(i) \in \texttt{featureCC}(f))$ **do**
  - $\varphi = \texttt{addSpecificFeature}(f, \varphi_{parent}, i)$
  - **foreach** $f' \in F$ *telle que* $(f, f') \in DE_{fm}$ **do**
    - $\texttt{FeatureTreeTraversal}(f', \varphi)$

mètres sont requis : une caractéristique $f \in F$ pour laquelle les CSFs correspondantes seront créées, et une CSF $\varphi_{parent} \in \Phi$ qui sera la CSF parent des CSFs nouvellement créées.

Les caractéristiques spécifiques au contexte sont créées différemment selon les trois situations suivantes :

1. La caractéristique f n'est associée à aucun concept ($\texttt{featureCC}(f) = \varnothing$) et la CSF parent φ_{parent} n'est associée à aucune instance ($\texttt{inst}(\varphi_{parent}) = \varnothing$).
   Alors, une CSF φ est créée. φ_{parent} est sa CSF parent. φ est associée à f et à aucune instance. La fonction addSpecificFeature met à jour le CSFM avec la nouvelle CSF. La procédure featureTreeTraversal est appelée récursivement sur chaque caractéristique fille de f.
2. La caractéristique fille, notée f, de la caractéristique associée à φ_{parent} est associée

à un concept ($featureCC(f) \neq \varnothing$), et la CSF parent φ_{parent} n'est pas associée à une instance ($inst(\varphi_{parent}) = \varnothing$).
Alors, pour chaque instance i du contexte étant instance d'un concept associé à f une CSF est créée en appelant la fonction addSpecificFeature. La CSF créée est associée à f et à une instance i. La procédure featureTreeTraversal est ensuite appelée récursivement pour chaque caractéristique fille de f afin de construire le sous arbre de chaque CSF φ précédemment créée.

3. La caractéristique fille, notée f, de la caractéristique associée à φ_{parent} est associée à un concept, et la CSF parent, notée φ_{parent}, est associée à une instance j.
Alors, pour toute instance i qui est une instance du concept associé à f ($A_C(i) \in featureCC(f)$), et qui est égale à l'instance associée à φ_{parent} ($i = inst(\varphi_{parent})$), satisfaisant la relation $jP_{N_I}i$, les actions suivantes sont effectuées :
   - Une CSF fille, associée à la caractéristique f et l'instance i, est ajoutée à φ_{parent}.
   - La procédure featureTreeTraversal est appelée récursivement pour chaque caractéristique fille de f afin de construire le sous-arbre de chaque CSF φ.

La fonction addSpecificFeature crée, et retourne, une nouvelle CSF dans le le CSFM. Elle a trois paramètres :

- la caractéristique f à laquelle est associée la nouvelle CSF,
- la CSF parent φ_{parent} de la nouvelle CSF,
- une instance i, facultative, à laquelle est associée la nouvelle CSF.

En premier lieu, la fonction crée une nouvelle CSF φ associée à i et à f. Sa multiplicité est calculée selon la définition 7. La borne inférieure correspond au maximum entre la borne inférieure de la multiplicité de la relation entre le concept dont i est l'instance et f, et la borne inférieure de la multiplicité de f. La borne supérieure correspond au minimum entre la borne supérieure de la multiplicité de la relation entre le concept dont i est l'instance et f, et la borne supérieure de la multiplicité de f.

Ensuite, la procédure addNewCSFtoGroup est appelée afin d'ajouter la nouvelle CSF à un groupe si f appartient à un groupe dans le FM. Si la nouvelle CSF est associée à une instance, elle doit aussi appartenir à un autre groupe dont la multiplicité est identique à celle de la caractéristique associée à la CSF. Ce groupe n'existe pas dans le FM. Son but est de transposer la multiplicité de la caractéristique f dans le CSFM afin d'assurer que le nombre de CSF pouvant être sélectionnées respecte la multiplicité de f. La CSF φ est ajoutée au groupe s'il correspond aux critères suivants :

- il groupe des CSF filles de φ_{parent},
- sa multiplicité est la même que celle de f,
- il groupe des CSF dont l'instance est associée au même concept que φ,
- il n'est associé à aucun groupe du FM,

Sinon un groupe est créé conforme aux critères précédemment énoncés, *i.e.*, tel que $g_{csfm} = (\varphi_{parent}, \lambda(f), \{\varphi\}, \varnothing)$.

La procédure addNewCSFtoGroup ajoute une CSF φ au groupe de sa CSF parent

Function addSpecificFeature(f, φ_{parent}, i) : $\varphi \in \Phi$

Input: Une caractéristique $f \in F$, une CSF $\varphi_{parent} \in \Phi$, et une instance de concept du contexte $i \in I$

Output: Met à jour le CSFM avec une nouvelle CSF, et retourne la CSF nouvellement créée.

```
// On crée une nouvelle CSF, on détermine sa multiplicité,
// et on l'ajoute à l'ensemble Φ des CSF.
```

Créer $\varphi = (f, i, (max(Min_M(featureCC_{mult}(featureCC(f))), Min_M(\lambda(f))),$
$min(Max_M(featureCC_{mult}(featureCC(f))), Max_M(\lambda(f)))))$

$\Phi = \Phi \cup \{\varphi\}$

$DE_{csfm} = DE_{csfm} \cup \{(\varphi_{parent}, \varphi)\}$

```
// On vérifie si φ doit être placé dans
// un groupe relatif à un groupe du FM.
addNewCSFtoGroup(φ, φ_parent)
```

if $i \neq \varnothing$ **then**

  if $\exists g_{csfm} \in G_{csfm} : g_{csfm} = (\varphi_{parent}, \lambda_{fm}(f), S_\varphi, \varnothing)$ *et* $S_\varphi \subset \Phi$ *et* $\forall \varphi_S \in S_\varphi, A_C(inst(\varphi_S)) = A_C(inst(\varphi))$ **then** `// Est-ce qu'un groupe de CSF existe ?`

    `// On ajoute la CSF au groupe.`

    $S_\varphi = S_\varphi \cup \varphi$

  else

    Créer $g_{csfm} = (\varphi_{parent}, \lambda(f), \{\varphi\}, \varnothing)$

    $G_{csfm} = G_{csfm} \cup \{g_{csfm}\}$

return φ

φ_{parent}. Premièrement, on vérifie si f appartient à un groupe dans le FM. Deuxièmement, s'il existe un groupe g auquel appartient f, alors on vérifie s'il existe déjà un groupe correspondant dans le CSFM. Le groupe doit :

- appartenir à φ_{parent},
- avoir la même multiplicité que g,
- être associé à g.

Si ce groupe existe, la CSF φ est ajoutée à l'ensemble S_φ, sinon le groupe est créé.

Procedure addNewCSFtoGroup($\varphi \in \Phi, \varphi_{parent} \in \Phi$)

Input: $\varphi \in \Phi$, CSF à ajouter à un groupe de φ_{parent}. $\varphi_{parent} \in \Phi$, CSF parent de φ.
Result: Ajoute la CSF fille φ à un groupe de la CSF φ_{parent} si nécessaire. Le groupe est créé s'il n'existe pas.

$f = feat(\varphi)$
$f_{parent} = feat(\varphi_{parent})$

if $\exists g \in G_{fm}$ *tel que* $g = (f_{parent}, (n,m), S)$, $S \subset F$, $f_{parent} \in F$, $(n,m) \in Mult$ *et* $\exists (f_{parent}, f) \in DE_{fm}$ *tel que* $f \in S$ *et* $f_{parent} \in F \backslash S$ **then** // Si f appartient à un groupe g.

  if $\exists g_{csfm} \in G_{csfm} : g_{csfm} = (\varphi_{parent}, \lambda_{G_{fm}}(g), S_\varphi, g)$ *et* $S_\varphi \subset \Phi$ **then** // S'il existe un groupe g_{csfm} appartenant à φ_{parent} relatif à g.

    // Alors φ doit appartenir au groupe g_{csfm}.
    $S_\varphi = S_\varphi \cup \{\varphi\}$
  else
    Créer $g_{csfm} = (\varphi_{parent}, \lambda_{G_{fm}}(g), \{\varphi\}, g)$
    $G_{csfm} = G_{csfm} \cup \{g_{csfm}\}$

La section suivante décrit la mise en œuvre de cette approche.

5.5 Implémentation

Nous avons implémenté la méthode d'adaptation de modèle de caractéristiques en Java avec le framework Eclipse RCP (Rich Client Platform). L'application permet de :

1. charger un modèle de caractéristiques fourni par l'utilisateur,
2. charger l'instance d'un modèle de contexte fourni par l'utilisateur,
3. générer le CSFM conformément au modèle de caractéristiques, à l'instance du modèle de contexte, et au modèle de contexte,
4. configurer le CSFM.

La figure 5.21 montre notre application Eclipse RCP lors de la phase de configuration du produit. La fenêtre est divisée en deux vues. La vue de gauche montre, sous la forme d'une arborescence, les instances du contexte ordonnées par le graphe N_I. La sélection de l'une de ces instances a pour effet de montrer dans la vue de droite un extrait du CSFM où toutes les CSF affichées sont associées à l'instance sélectionnée à gauche.

La sélection de la première instance de la vue de gauche “Business-model independent features” permet de présenter à droite un extrait du CSFM montrant les CSF qui ne sont associées à aucune instance.

Si l’instance associée à une CSF ne permet pas de satisfaire la contrainte de la CSF, alors un message d’erreur informe l’utilisateur de l’incompatibilité de l’instance avec la caractéristique. Un autre choix ergonomique possible consisterait à ne présenter que des CSF sélectionnables en analysant au préalable l’instance à laquelle elles sont associées.

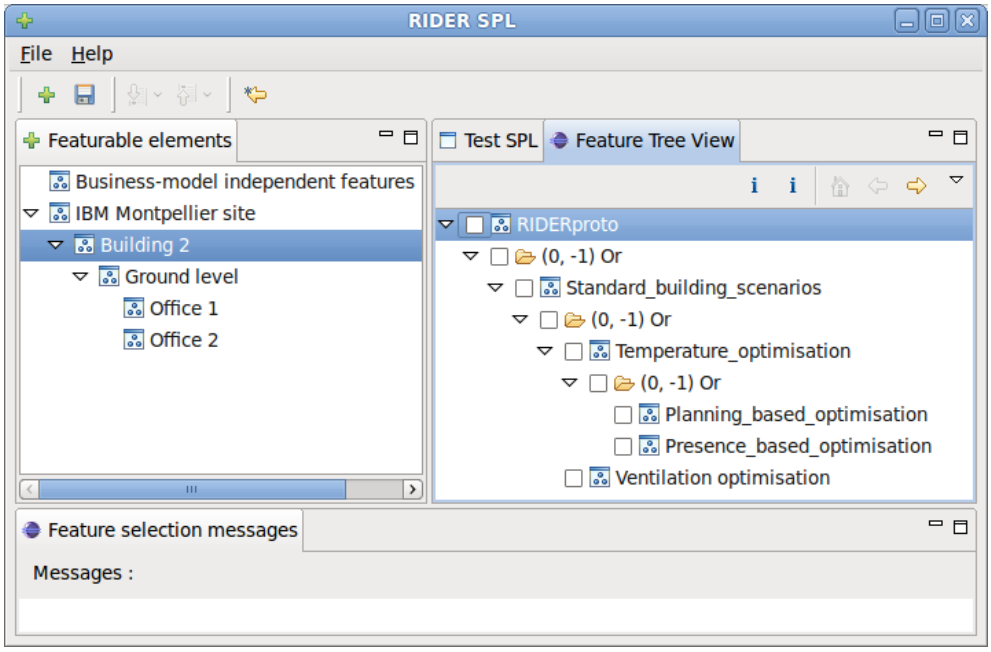

FIGURE 5.21 : Capture d’écran du configurateur avec le CSFM d’un logiciel d’optimisation énergétique de bâtiments

L’approche est générique car notre application est indépendante des modèles utilisés (le CM, le CMI, et le FM). Si le contexte change, la transformation doit utiliser une nouvelle instance du modèle de contexte représentant le nouveau contexte. Seul ce modèle doit changer. Si le modèle de contexte change, dans la version actuelle du prototype, il faut générer les classes Java du modèle de contexte à partir du schéma XML utilisé pour décrire les instances de ce modèle. Il faut également générer le modèle représentant la variabilité à partir des stéréotypes placés sur le modèle UML du modèle de contexte (décrit dans le chapitre 4, figure 4.17). De façon générale, lorsqu’on crée une nouvelle ligne de produits, tous les modèles sont différents, mais la transformation reste identique.

5.6 Conclusion

Dans ce chapitre, nous avons présenté une approche permettant d’adapter statiquement des modèles de caractéristiques génériques au contexte d’exécution du futur produit.

Le modèle de caractéristiques adapté au contexte est défini à partir d'une instance de modèle de contexte, et d'un modèle de caractéristiques générique. Un algorithme met en œuvre cette approche et duplique si possible chaque caractéristique en analysant les éléments du contexte selon des contraintes définies dans le modèle de caractéristiques générique. Ainsi, à chaque instance d'un modèle de contexte correspond un unique modèle de caractéristiques adapté dans lequel chaque caractéristique dupliquée est associée à une instance du contexte.

Notre méthodologie a été conçue de façon générique afin d'être réutilisée facilement dans d'autres domaines. Nous avons développé une application basée sur le framework Eclipse RCP capable de générer un modèle de caractéristiques adapté et de fournir à l'utilisateur une interface graphique permettant de le configurer.

Une première perspective future consisterait à définir les conditions sous lesquelles un triplet FM, CM et CMI peut donner naissance à un CSFM valide (par exemple, multiplicités compatibles, présence nécessaire de certaines instances si des caractéristiques sont obligatoires).

Autre perspective future, la présentation des vues sur le CSFM permettrait de simplifier la sélection de caractéristiques dans le cas de modèles très grands. Nos vues permettent déjà de filtrer les CSF affichées, mais nous pourrions également utiliser des vues spéciales permettant d'afficher et de sélectionner simultanément un ensemble de CSF associées à une même instance du contexte. Une autre application de ces vues consisterait à permettre la configuration d'un sous ensemble de CSFM dont toutes les CSF se rapportent à une même instance du contexte. Ainsi, la configuration d'un grand nombre de CSF se rapportant à un même concept pourrait être réalisée manuellement.

Un autre usage des vues permettrait de ne présenter que les CSF relatives au point de vue de l'un des commanditaires du produit. Par exemple, les CSF relatives au fonctionnement technique du logiciel ne seraient pas présentées à une personne chargée de choisir les fonctionnalités haut niveau du logiciel.

Une autre perspective consiste à automatiser la configuration d'un CSFM selon des critères permettant de réaliser la configuration la "meilleure" possible. Le choix entre plusieurs caractéristiques pourrait être guidé par une pondération de ces caractéristiques afin de déterminer le meilleur choix possible. Par exemple, le choix des logiciels pourrait être effectué en fonction du volume des données à traiter, estimé à partir de la taille du bâtiment et du nombre de capteurs installés. Cela permettrait d'installer des logiciels plus légers, et nécessitant un ordinateur moins puissant.

Une évolution possible consisterait à permettre la configuration d'un nouveau produit en fonction des caractéristiques présentes dans un écosystème de produits existants, dans le but de les rendre inter-opérationnels. Par exemple, dans le cas où plusieurs produits sont amenés à être coordonnés ensemble, comme dans un contexte multi-agents, des rôles pourraient être répartis entre les différents produits existants.

Une dernière perspective possible consisterait à permettre la reconfiguration dyna-

mique d'un produit. Actuellement, modifier un produit requiert la reconfiguration et le re-déploiement complet du produit.

CHAPITRE

6 Application du méta-modèle et des algorithmes proposés dans le contexte du projet RIDER

Sommaire

6.1 Introduction **97**
6.2 Modèle de caractéristiques générique de RIDER **99**
6.3 Modélisation du bâtiment **110**
6.4 Modèle de caractéristiques adapté à un bâtiment **115**
6.5 Conclusion **118**

6.1 Introduction

Ce chapitre décrit un exemple de mise en œuvre de notre nouveau méta-modèle de caractéristiques (cf. chapitre 4) et de l'algorithme d'adaptation au contexte de modèles de caractéristiques (cf. chapitre 5) dans la création de la ligne de produits du projet RIDER. L'idée est de montrer les différentes étapes permettant de présenter à la personne qui configure un nouveau produit un modèle de caractéristiques adapté au contexte de son bâtiment. La configuration effectuée par cette personne permettra la création et le déploiement d'une instance de RIDER. Pour ce faire, il est nécessaires d'utiliser les modèles décrits dans la figure 6.1. Chacun d'eux est créé, puis réutilisé, à une étape précise de la création de la ligne de produits selon notre méthode :

1. Création du méta-modèle du contexte (le *Context Model* CM [1]) géré par les produits. Il s'agit dans le contexte de RIDER du méta-modèle d'infrastructure présenté au chapitre 2. Il a été réalisé en UML avec Rational Software Architect (RSA).
2. Construction du modèle générique de caractéristiques (FM) représentant l'ensemble des produits de la ligne de produits. Les caractéristiques qui représentent un composant logiciel concret doivent être associées à un identifiant de ce composant dans un

1. Les abréviations CM, FM, *etc.* sont celles définies au chapitre 5.

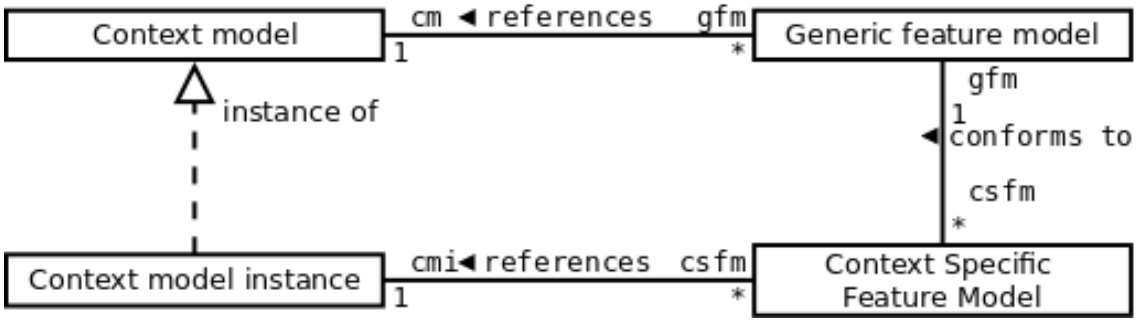

FIGURE 6.1 : Vue d'ensemble des modèles de notre approche

référentiel réalisé à l'aide de la norme *Reusable Asset Specification* (RAS) [OMG, 2005]. RSA supporte cette norme et permet de gérer un référentiel et les assets qu'il contient. Les caractéristiques dont le nombre d'occurrences dépend de la présence d'instances d'un concept sont associées aux concepts du CM créé précédemment. Dans notre contexte, la construction du FM a été réalisée avec RSA à l'aide du profil UML présenté au chapitre 4.

3. Instanciation du CM (création du *Context Model Instance* CMI) pour modéliser le bâtiment (la zone pilote du bâtiment IBM) pour lequel on souhaite réaliser une instance de RIDER. La section 6.3 décrit le CMI de la zone pilote du bâtiment IBM. Nous avons illustré cette étape en modélisant le CMI sous la forme de diagrammes d'instances UML.
4. Génération du modèle de caractéristiques spécifique au contexte du bâtiment IBM. Cette étape est automatisée par l'algorithme de génération décrit au chapitre 5. La section 6.4 présente le CSFM, généré par l'algorithme, de la zone pilote du bâtiment IBM.
5. Sélection des caractéristiques du CSFM pertinentes pour les commanditaires du logiciel. La section 6.4 illustre également ce processus.

Le projet RIDER en est à la phase de finalisation d'une première architecture sur le premier bâtiment pilote. Le développement et la spécification de la ligne de produits proprement dite sont encore à faire. L'architecture actuelle est constituée de composants logiciels de grosse granularité dont la configuration et le paramétrage sont limités. Cela ne permet donc que peu de variabilité. Notre étude prospective a pris comme exemple les caractéristiques de l'architecture actuelle auxquelles nous avons ajouté des caractéristiques prévues pour les développements futurs. Les modules suivants seront mis en place à terme :

- Visualisation 3D de bâtiments,
- Recommandation multicritères pour l'efficacité énergétique utilisant une modélisation du confort des habitants,
- Système d'auto apprentissage et algorithme d'aide à la décision pour l'optimisation énergétique.

Ce chapitre donne un exemple d'application de nos modèles et de l'algorithme d'adaptation au contexte pour l'une des multiples instanciations prévues de RIDER. Au total, trois sites pilotes, de nature et d'usages très différents, sont prévus :

1. PSSC IBM Montpellier : Centre de calcul construit en 1964 composé d'une zone de bureaux et d'un green data center. D'importantes pertes énergétiques sont subis à cause des matériaux et techniques de construction utilisées à l'époque. Cependant, d'importants travaux ont été réalisés afin de permettre des mesures et un contrôle très précis des systèmes de chauffage et climatisation, des améliorations en matière d'isolation thermique, et un système de réutilisation de l'énergie fatale issue des data centers.
2. Le théâtre de l'archipel à Perpignan : Théâtre moderne achevé en 2012, d'une capacité de 1200 places, équipé d'un système géothermique. Ce bâtiment bénéficie d'une excellente isolation thermique et permet également des mesures et un contrôle précis de l'infrastructure.
3. Polytech Perpignan : Plusieurs bâtiments récents, composés de salles de cours et d'amphithéâtres. Des capteurs supplémentaires sont installés afin de permettre des mesures et un contrôle similaire aux deux autres sites.

L'exemple suivi dans ce chapitre concerne le premier site, le PSSC IBM Montpellier.

6.2 Modèle de caractéristiques générique de RIDER – FM

Le tableau 6.1 présente les principales caractéristiques relatives à l'architecture de RIDER et les différents choix possibles de composants logiciels pour les implémenter. Certains des composants logiciels sont basés sur des logiciels qui doivent être programmés ou configurés pour RIDER (*SPSS Modeler*, *R*, *TRNSys*), et sur des logiciels développés spécifiquement pour RIDER.

Dans la pratique, certains composants logiciels tels que les systèmes de gestion technique centralisée de bâtiments (GTC) sont utilisés en tant que "boîtes noires" et paramétrés de façon ad-hoc par le fournisseurs du logiciel (Pyrescom, Enoleo, Synapsense). Ainsi, il est possible d'effectuer un choix parmi ces composants, mais il n'est actuellement pas possible d'en configurer le fonctionnement.

Caractéristiques	Choix possibles de composants logiciels
Optimisation températures par réseau de neurones	SPSS Modeler, Matlab, R, développement RIDER
Visualisation	Développement RIDER Visu 3D, développement RIDER Visu 2D
Recommandation multicritères	Matlab, R, développement RIDER
Système d'auto apprentissage et algorithme d'aide à la décision	Matlab, R, TRNSys, développement RIDER
Dashboards	Cognos, Pyrescom Oracle Business Intelligence,
Stockage historique et data warehouse	DB2, Oracle DB, PostgreSQL
Interface superviseurs	Enoleo, Pyrescom, Synapsense Interface RIDER, Johnson Controls, Schneider Electric
Reporting et alertes	Tivoli Netcool/OMNIbus, Pyrescom
Instrumentation du bâtiment	Pyrescom, Enoleo, Synapsense

TABLE 6.1 : Présentation des principales fonctionnalités de RIDER et de choix de composants logiciels pour les implémenter

Dans le chapitre 2, nous avons décrit l'architecture de RIDER dont nous présentons les principales couches [Brad Brech *et al.*, 2011] dans la figure 6.2 ci-dessous. Cette architecture permet l'interconnexion de RIDER avec un système de gestion technique centralisée (GTC, ou Building Management System – BMS) de bâtiments. Les caractéristiques sont organisées dans des couches (*Layers*) représentées par des rectangles blancs. Ces couches représentent les couches de l'architecture IBM. Pour l'exemple, nous avons inclus les caractéristiques relatives à la configuration des logiciels concernant les GTC du bâtiment et ses instruments de mesure et de contrôle. Ces caractéristiques sont dans les couches *buildingManagementSystems* et *instrumentationSystems* présentées dans le rectangle bleu.

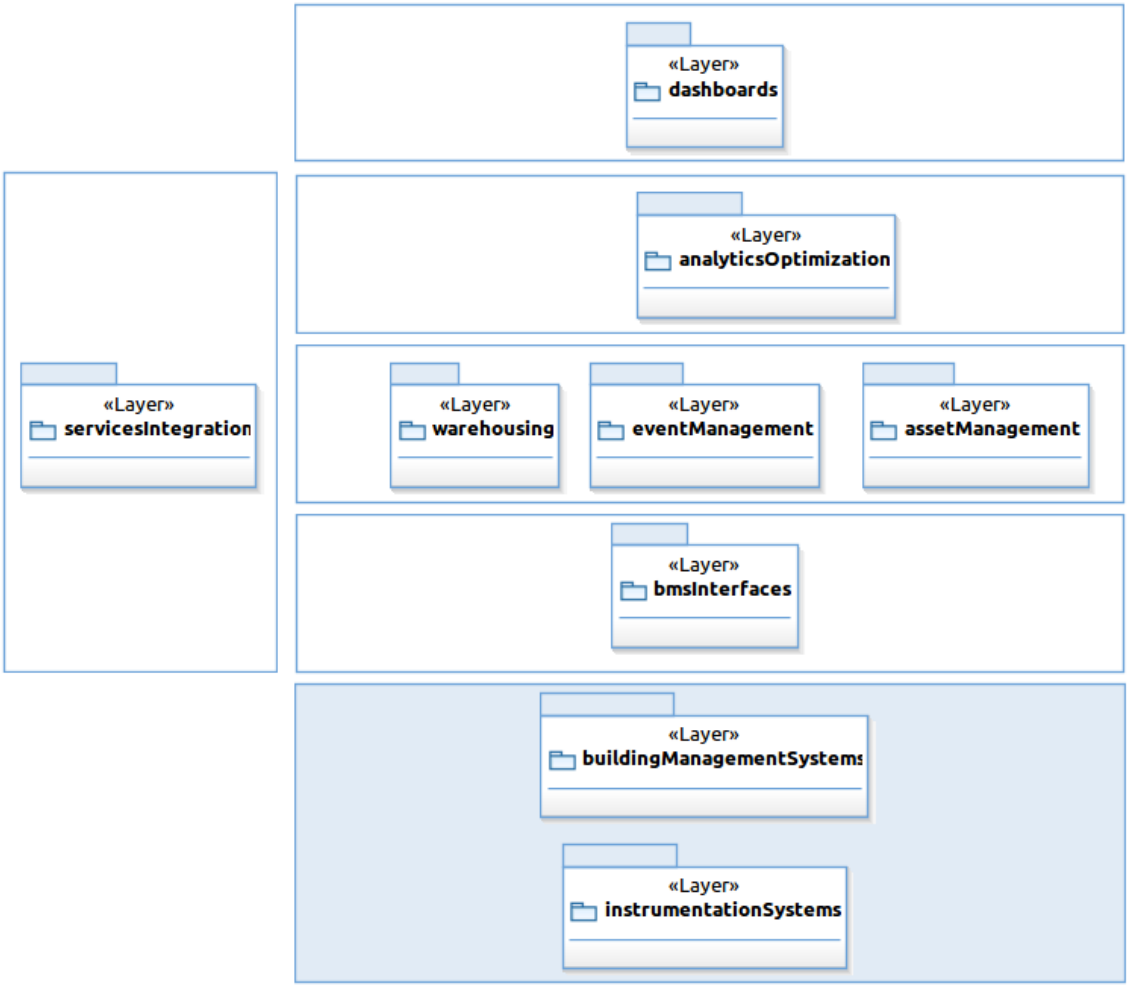

FIGURE 6.2 : Couches organisant le modèle de caractéristiques générique (noté FM) de RIDER

- La couche *bmsInterfaces* contient les différentes caractéristiques, montrées dans la figure 6.3 permettant l'interconnexion entre un produit RIDER et les GTC d'un bâtiment. Chacune des caractéristiques représente une interface particulière (par exemple, *Enoleo*, *Synapsense*). Elles ont la multiplicité *1* car une seule occurrence du composant logiciel représenté par chacune de ces caractéristiques est nécessaire.

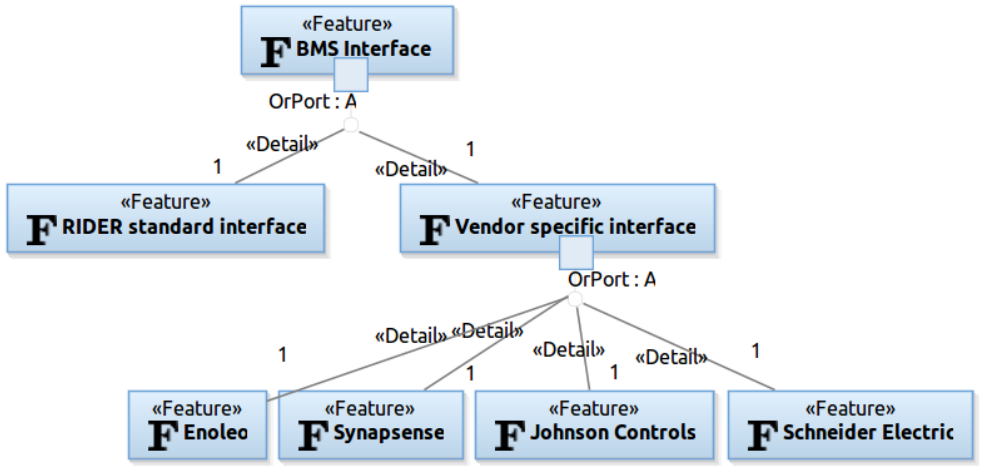

FIGURE 6.3 : Caractéristiques des interfaces de communication entre RIDER et les GTC

- Les couches *warehousing*, *eventManagement* et *assetManagement* présentent trois différents types de traitement de données. Les caractéristiques correspondantes sont sont détaillées respectivement dans les figures 6.4, 6.5, et 6.6.
  - La couche *warehousing* contient les caractéristiques, décrites dans la figure 6.4, représentant les composants logiciels liés à l'agrégation des données issues de différentes GTC et à leur stockage dans un data warehouse. Cette couche traite les données mesurées notamment par les capteurs et les compteurs.
    Trois caractéristiques principales sont nécessaires :

    1. *Data aggregation*,
    2. *ETL*,
    3. *Data warehouse*.

    Chacune elles peut être implémentée d'une manière spécifique. Par exemple, la caractéristique *Data aggregation* peut être implémentée par les caractéristiques *SPSS aggregation webservice* ou *DB2 webservice*. Le choix de la caractéristique *SPSS aggregation webservice* requiert également le choix de la caractéristique *SPSS transformation* qui implémente la caractéristique *ETL*.

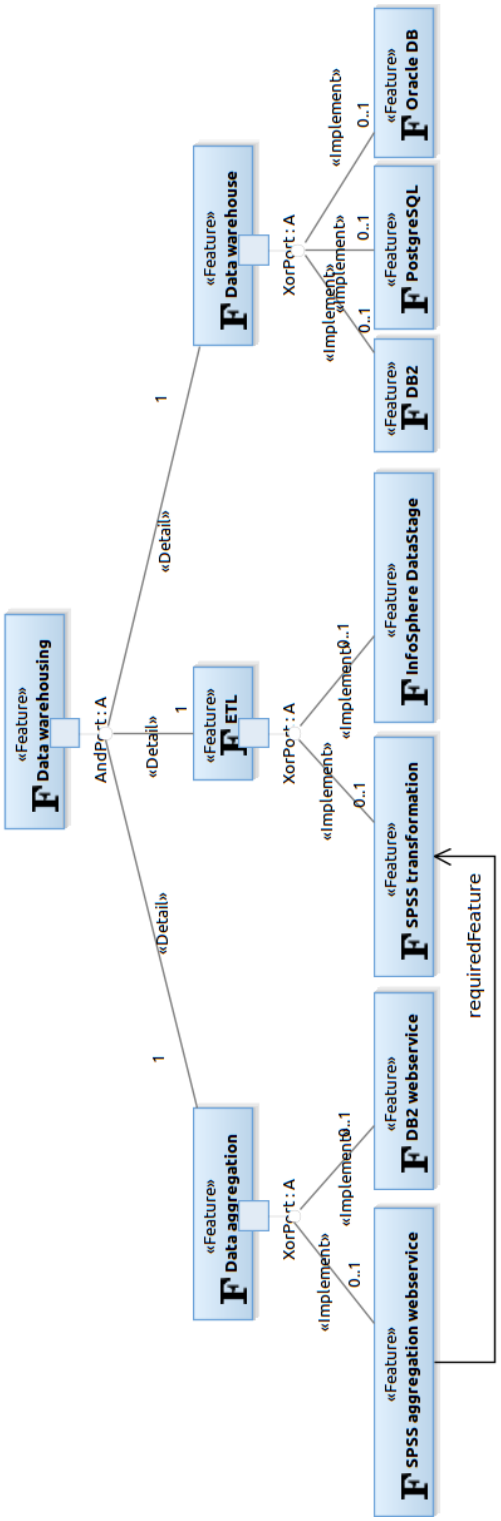

FIGURE 6.4 : Caractéristiques liées à la mise en œuvre d'un data warehouse

– La couche *eventManagement* contient les caractéristiques relatives aux systèmes effectuant le traitement des événements envoyés par les GTC. Les événements traités concernent par exemple les défaillances de capteurs ou erreurs de fonctionnement.

Cela permet notamment de déterminer quand effectuer des opérations de maintenance sur des équipements. La figure 6.5 décrit les caractéristiques représentant les deux composants logiciels utilisables pour réaliser cette tâche.

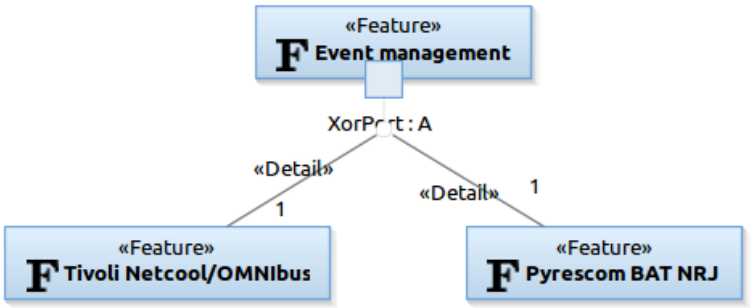

FIGURE 6.5 : Caractéristiques liées aux différents systèmes de gestion d'événements

– La couche *assetManagement* décrit les caractéristiques, illustrées dans la figure 6.6 relatives aux systèmes de gestion d'actifs. Ils effectuent la gestion des équipements et de l'occupation des lieux. Ils servent par exemple, à superviser la maintenance d'équipements en consignant des informations techniques, l'historique des interventions, la date d'installation, la garantie, et les vendeurs de l'équipement. D'autres alternatives à la caractéristique *Maximo Asset Management* pourront venir compléter la ligne de produits dans le futur.

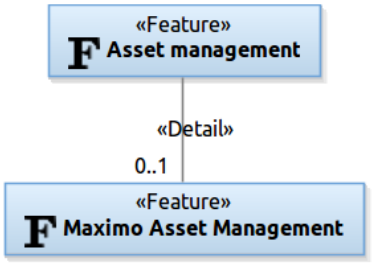

FIGURE 6.6 : Caractéristiques liées aux différents systèmes de gestion du patrimoine

- La couche *analyticsOptimization*, décrite dans la figure 6.7, contient les caractéristiques décrivant les fonctionnalités des outils d'analyse et d'optimisation intégrés à RIDER. Ces outils permettent par exemple, d'effectuer des rapports sur l'efficacité d'un équipement, de comparer l'efficacité de différents équipements entre eux, d'effectuer de l'analyse prédictive afin de détecter des patterns et d'anticiper des événements, de proposer des consignes permettant de faire des économies d'énergie.

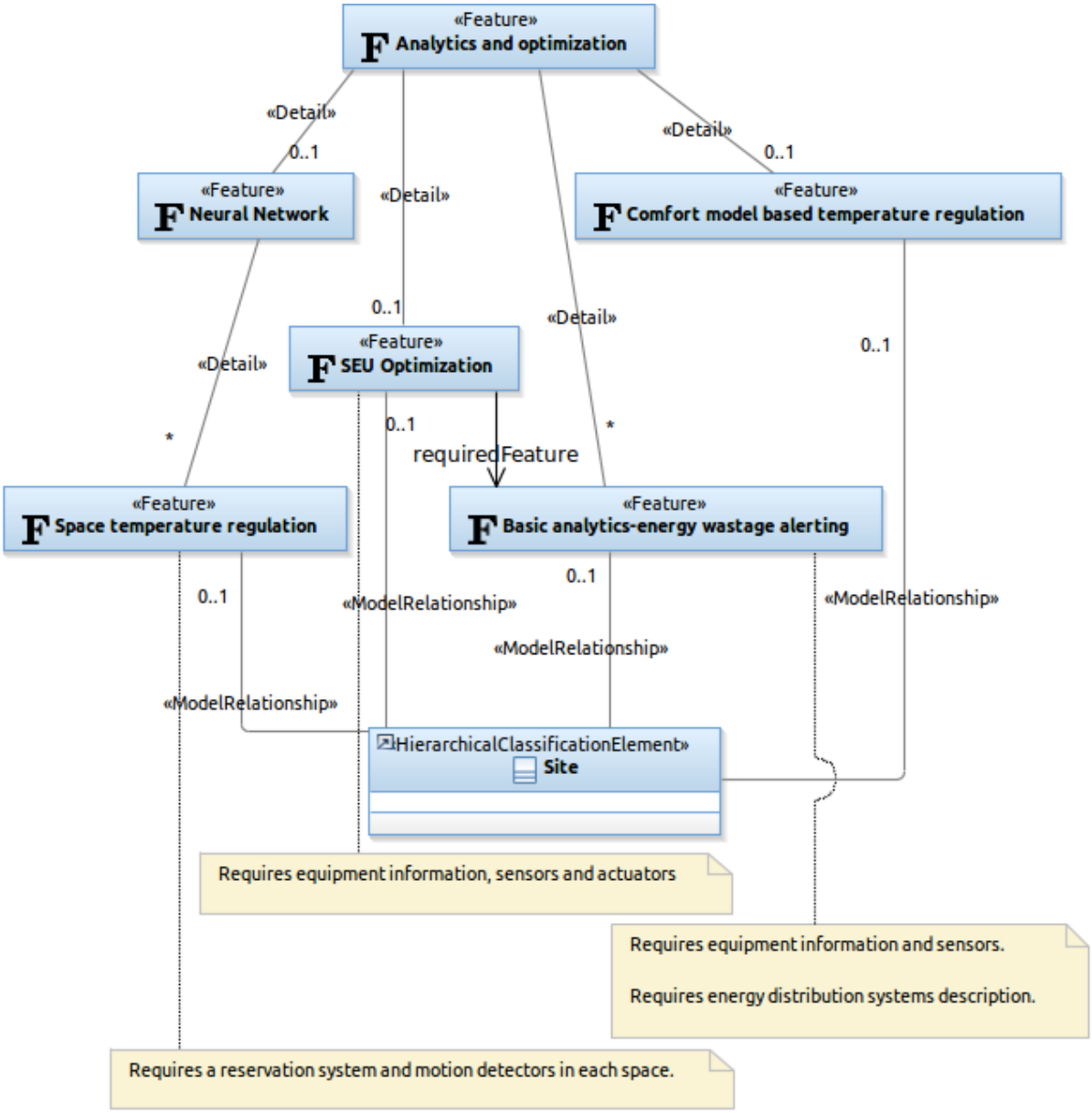

FIGURE 6.7 : Caractéristiques des différents types d'analyses et d'optimisations

La caractéristique *Neural Network* représente la présence d'une analyse par réseau de neurones dans l'architecture RIDER. Elle est détaillée par la caractéristique *Space temperature regulation* décrivant une fonctionnalité de régulation de la température des espaces du bâtiments. Cette caractéristique peut être présente 0 ou autant de fois qu'il y a de pièces, comme le dénotent sa multiplicité $*$ et son association au concept *Site*. La multiplicité $(0,1)$ de l'association *ModelRelationship* détermine que cette caractéristique peut être présente zéro ou une fois dans chaque espace (instance de *Site*).

- La couche *dashboards* contient les caractéristiques relatives aux différents dashboards permettant d'afficher des informations issues, par exemple, du data warehouse, et des outils d'analyse. La figure 6.8 présente les caractéristiques des différents dashboads disponibles.

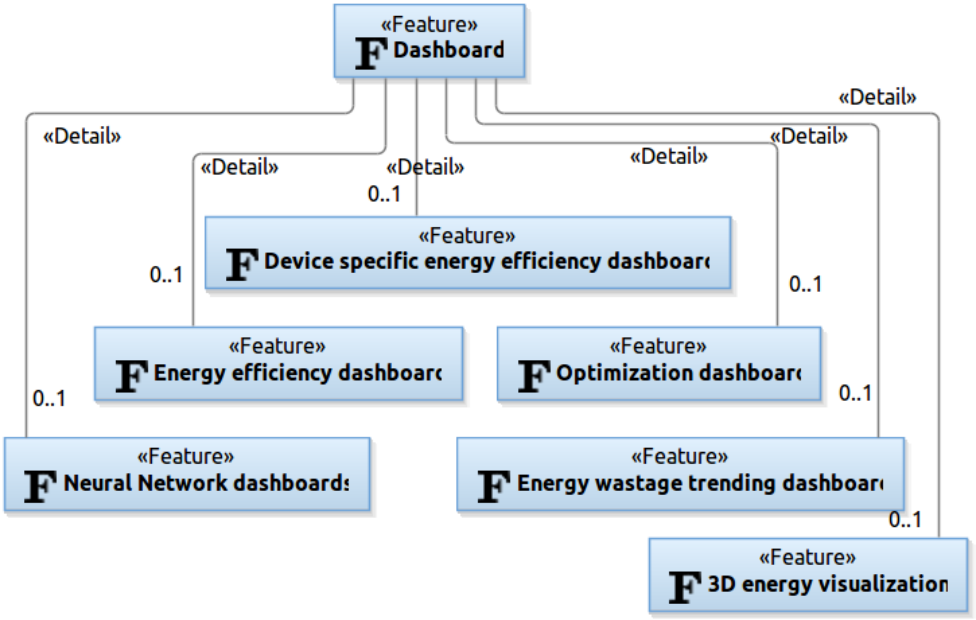

FIGURE 6.8 : Caractéristiques des différents dashboards

La figure 6.9 présente les contraintes de sélection entre les caractéristiques relatives aux outils d'analyse et aux tableaux de bords. Chaque outil nécessite la présence d'un ou plusieurs dashboards. Par exemple, la caractéristique *Neural Network* requiert la caractéristique *Neural network dashboads*.

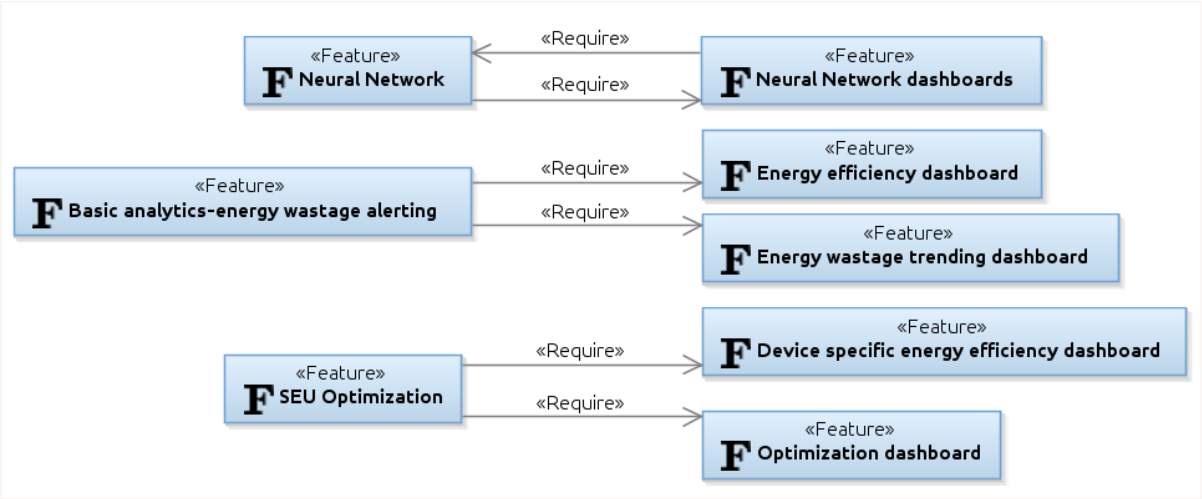

FIGURE 6.9 : Contraintes entre les caractéristiques de l'analyse de données et des dashboards

La figure 6.10 décrit les contraintes de sélection entre les caractéristiques relatives aux méthodes d'optimisation énergétiques et les caractéristiques représentant les composants logiciels permettant de mettre en œuvre ces méthodes d'optimisation.

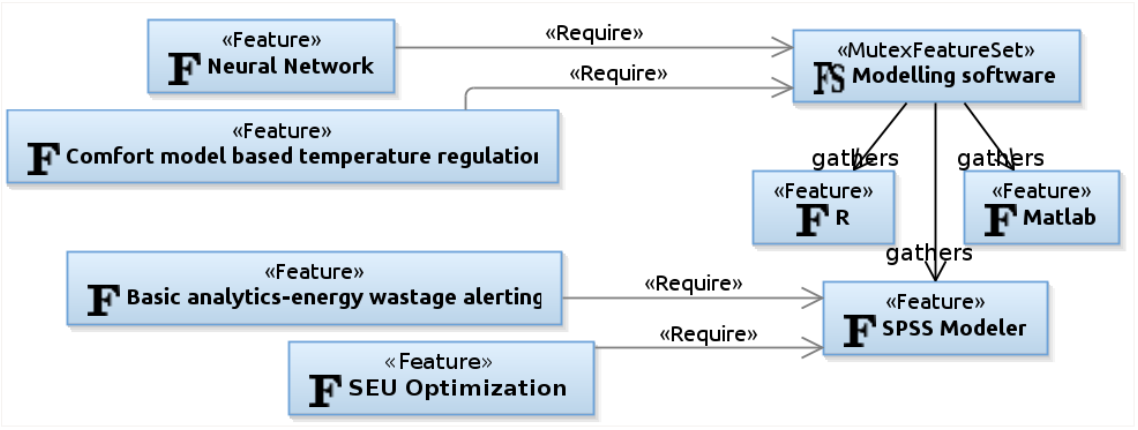

FIGURE 6.10 : Contraintes de sélection entre les caractéristiques d'optimisation et les composants logiciels

- La couche *servicesIntegration* contient les caractéristiques concernant l'intégration de services extérieurs au logiciel d'optimisation énergétique. La figure 6.11 décrit les caractéristiques disponibles. Par exemple, la caractéristique *Room schedule interface* représente l'intégration d'un service permettant d'acquérir l'emploi du temps d'une pièce. Ses deux caractéristiques filles décrivent respectivement des services d'acquisition du planning à partir d'un fichier Excel ou d'un planning du logiciel Lotus Notes. Ces logiciels ne font pas partie de RIDER, mais sont accessibles tels des services extérieurs.

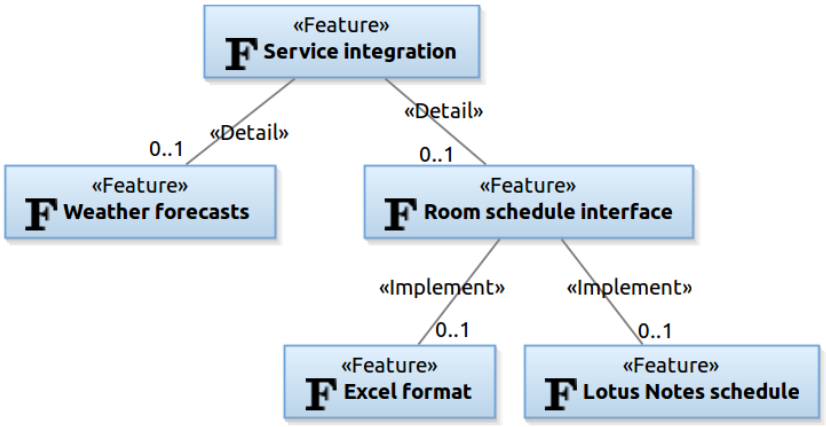

FIGURE 6.11 : Intégration de services extérieurs au logiciel d'optimisation énergétique

- La couche *buildingManagementSystems* contient les caractéristiques concernant les GTC utilisées dans le bâtiment. La figure 6.12 représente les caractéristiques relatives aux superviseurs utilisables dans un bâtiment dont la consommation énergétique est optimisée par RIDER.
  Chaque espace du bâtiment peut être supervisé par une GTC. Cela est représenté par l'association *ModelRelationship* entre le concept *Space* et la caractéristique *Building Management System*. Ainsi, chaque espace peut être supervisé par l'un des supervi-

seurs disponibles fourni par les sociétés Enoleo, Pyrescom, et Synapsense. Ceci est indiqué par le groupe “Xor” de la caractéristique *BMS provider*.
Les caractéristiques *Instrumented Space* et *Data center* déterminent si la GTC doit superviser un espace instrumenté tel qu'un bureau ou une salle de réunion, ou un data center. Dans le cas d'un data center, ce diagramme indique que seule la GTC fournie par Synapsense pourra le superviser car c'est la seule qui ne soit pas en conflit avec la caractéristique *Data center*. Les caractéristiques filles de ces deux caractéristiques décrivent les types de capteurs qui devront être gérés par la GTC.

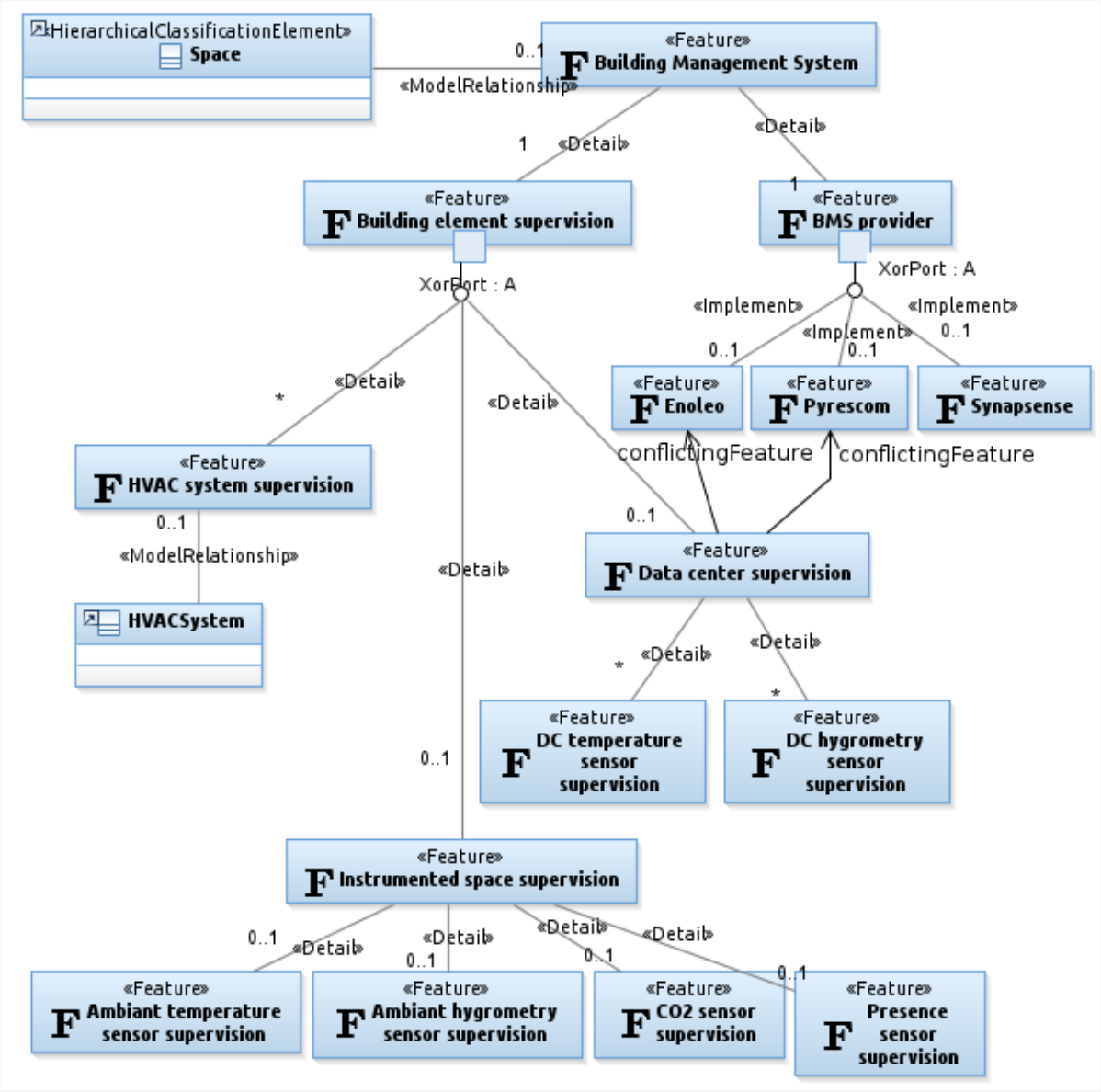

FIGURE 6.12 : Caractéristiques liées aux différents BMS

- La couche *InstrumentationSystems* décrit les caractéristiques de l'instrumentation du bâtiment. Certaines zones du bâtiment peuvent avoir des systèmes d'air conditionné, ou un système de refroidissement de data center (à l'aide d'air et d'eau). L'association

ModelRelationship entre le concept *SpatialZone* et la caractéristique *Zone instrumentation* détermine que toute zone indépendamment des autres peut avoir un système d'air conditionné. Les caractéristiques *HVAC system* et *DC cooling system* contiennent des règles, non présentées ici, permettant de déterminer si les instruments associés à une instance donnée du concept *SpatialZone* sont adéquat par rapport aux besoins d'instrumentation de RIDER.
L'association *ModelRelationship* entre le concept *Space* et la caractéristique *Space instrumentation* détermine que chaque espace dispose de son instrumentation. Il peut y avoir un ensemble de capteurs (lumière, CO_2, présence, hygrométrie, ou température) décrits par les caractéristiques filles de *Sensors*. Il peut également y avoir un ensemble d'actionneurs permettant de gérer l'air conditionné, décrits par les caractéristiques *Air conditionning*, *Damper*, et *Fan*.

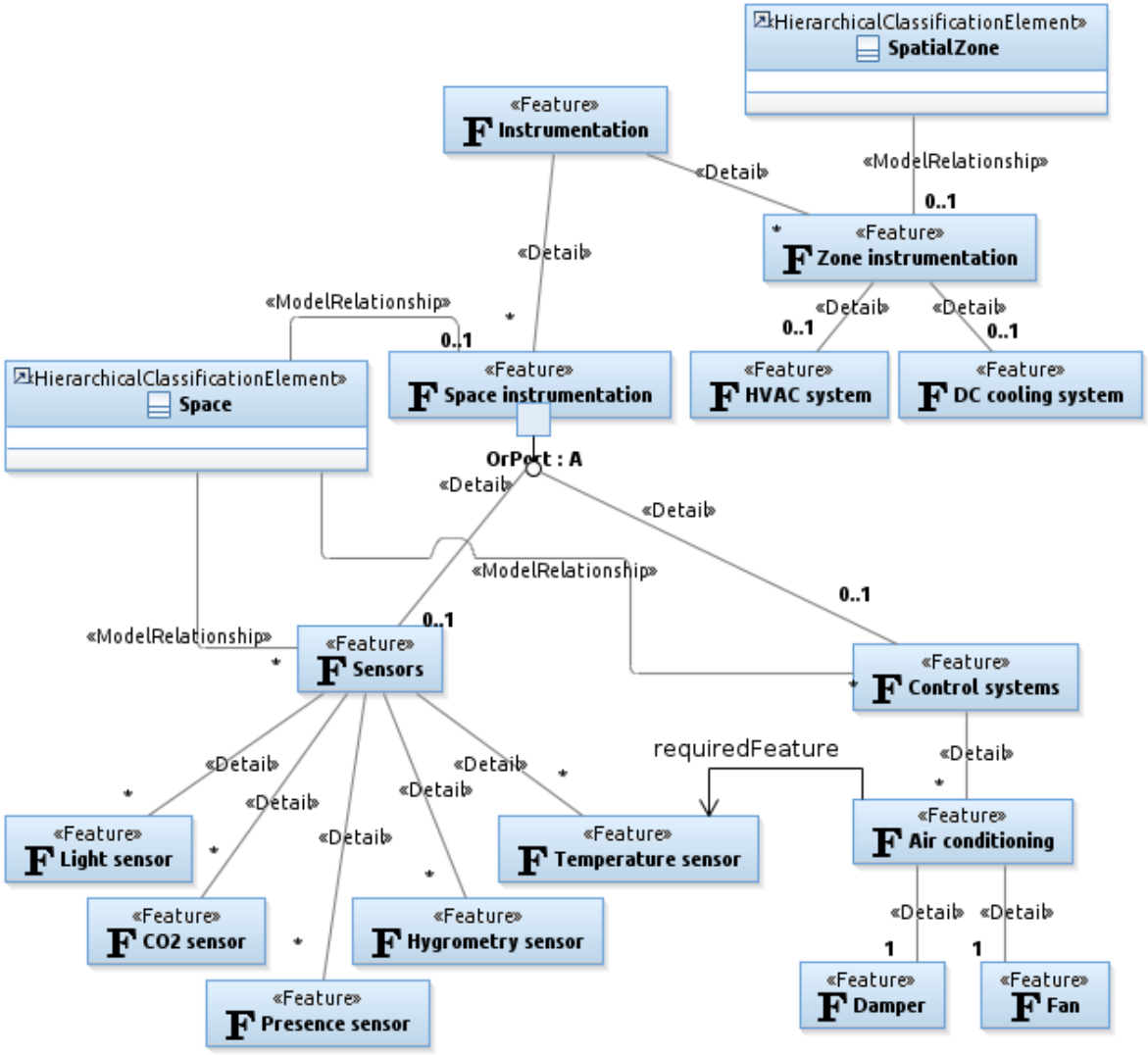

FIGURE 6.13 : Caractéristiques liées à l'instrumentation du bâtiment

La section suivante présente un exemple de CMI, instance du CM présenté dans le chapitre 2.

6.3 Modélisation du bâtiment – CMI

Cette section présente la manière dont nous avons instancié le modèle de bâtiment présenté au chapitre 2 pour modéliser la zone pilote du projet. Nous utilisons les diagrammes d'objets du langage UML pour représenter l'instance du modèle de bâtiments (réalisé à l'aide de diagrammes de classes).

La figure 6.14 présente le plan de la zone instrumentée ainsi que les capteurs présents dans chaque pièce. Des mesures de ces capteurs sont également montrées car il s'agit d'une capture d'écran issue du superviseur en fonctionnement contrôlant la zone[2]. Cette zone englobe une zone de bureaux, des salles de réunion, des salles de cours, et un hall.

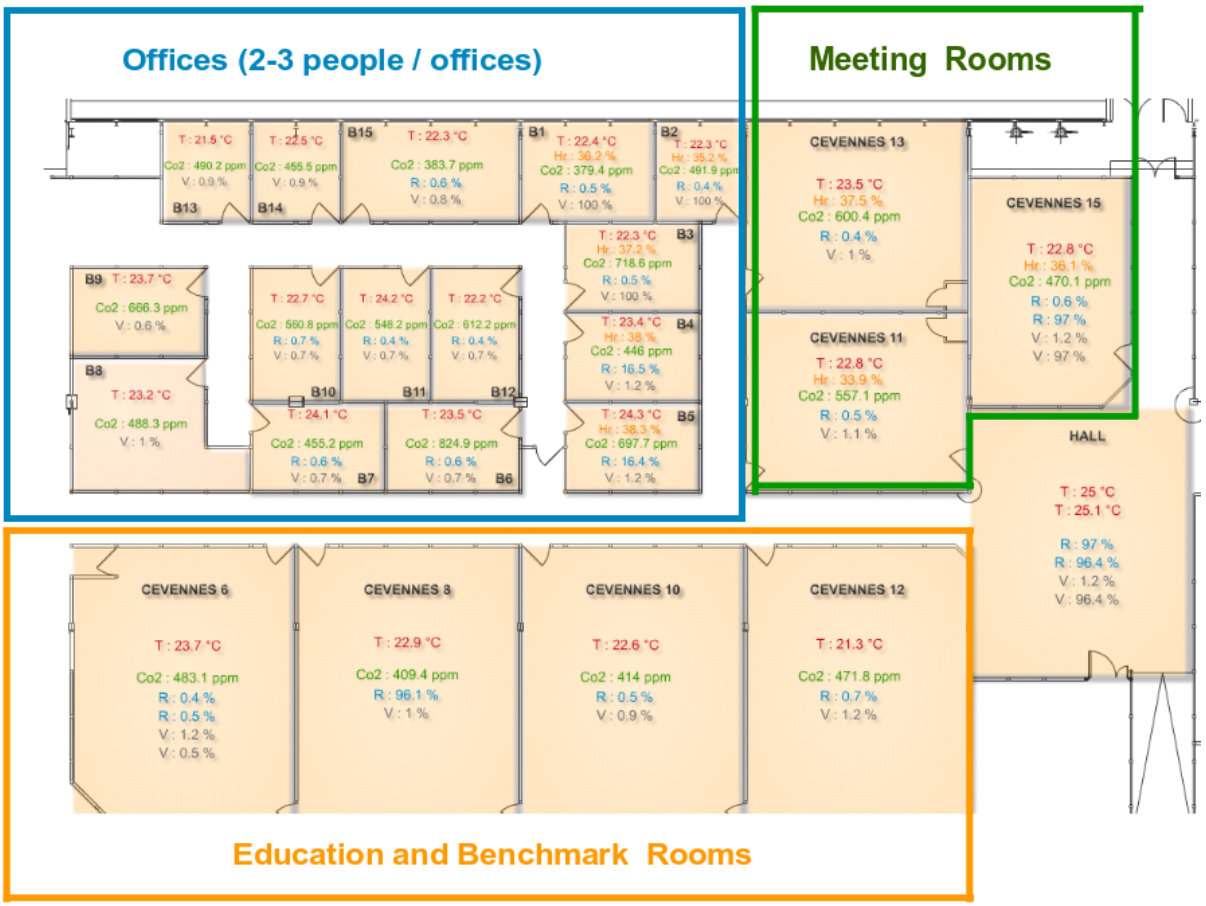

FIGURE 6.14 : Plan de la zone optimisée par RIDER

Dans ce cadre, le logiciel d'optimisation énergétique propose des consignes de température et de ventilation optimisées en fonction des informations reçues concernant l'occupa-

2. fourni par l'entreprise Enoleo

tion des salles et météorologiques. La GTC qui contrôle la zone applique ensuite au mieux ces recommandations selon les normes de sécurité en vigueur et les possibilités des équipements.

La figure 6.15 présente un diagramme d'objets réalisé à partir du plan de la zone. De plus, l'objet *zoneOcupation : Planning* indique que les salles de benchmark et de réunion possèdent un planning d'occupation.

Nous avons également représenté deux espaces qui modélisent les zones instrumentées du green data center. En effet, ce centre de calcul a fait l'objet de modifications spéciales permettant de réutiliser la chaleur produite par les serveurs pour alimenter le circuit d'eau chaude utilisé pour chauffer les bureaux de la zone pilote.

La figure 6.16 présente un exemple de modélisation d'une centrale de traitement d'air (CTA) gérée par l'un des superviseurs connectés à RIDER.

La CTA est divisée en trois parties reliées par l'objet *TPipe* de type *DistributionLink*. Cet objet représente un conduit d'air en forme de "T" permettant de mélanger de l'air extérieur à de l'air repris dans le bâtiment afin de le mettre à bonne température et le souffler dans le bâtiment. Ce conduit est instrumenté par deux clapets motorisés ("Dampers") représentés par les objets *RM100* et *RM200*. Ces clapets servent à réguler l'air issu des systèmes de récupération d'air neuf et d'air vicié.

Les trois parties sont représentées par des rectangles colorés :

- Le rectangle jaune contient les objets relatifs à récupération d'air vicié. L'objet *IndoorRecupAir* représente la bouche de récupération d'air dans un local. Plusieurs capteurs permettent de mesurer la température, l'hygrométrie et la vitesse du flux d'air. Cette arrivée d'air est connectée à un conduit d'aération en forme de "T", représenté par l'objet *RecupAirTPipe*, permettant de doser la quantité d'air qui sera réutilisée dans le bâtiment ou rejetée à l'extérieur.
- Le rectangle bleu représente le conduit d'arrivée d'air neuf. Il est instrumenté par des capteurs de température et d'hygrométrie.
- Le rectangle à bord rouge représente la CTA proprement dite. La CTA est divisée en trois batteries de traitement d'air. La batterie A est équipée de capteurs de température, d'hygrométrie et de vitesse de flux d'air. La batterie B est un filtre permettant de nettoyer l'air. La batterie C contient deux échangeurs thermiques alimentés par des réseaux d'eau froide et chaude. Ils ne sont pas représentés sur ce schéma par souci de clarté. L'objet *BlowedAir* représente la sortie de la CTA, raccordée au système de distribution d'air neuf du bâtiment.

L'ensemble des informations issues de ces instruments sont utilisés par les algorithmes d'optimisation énergétique. En retour, les algorithmes peuvent proposer des valeurs de consignes à respecter. Par exemple, une consigne peut concerner le pourcentage d'air neuf à utiliser, ou la température et le débit de l'air soufflé le bâtiment.

Nous ne décrivons pas dans cet exemple l'ensemble des systèmes de distribution d'air et d'eau. Il manque notamment :

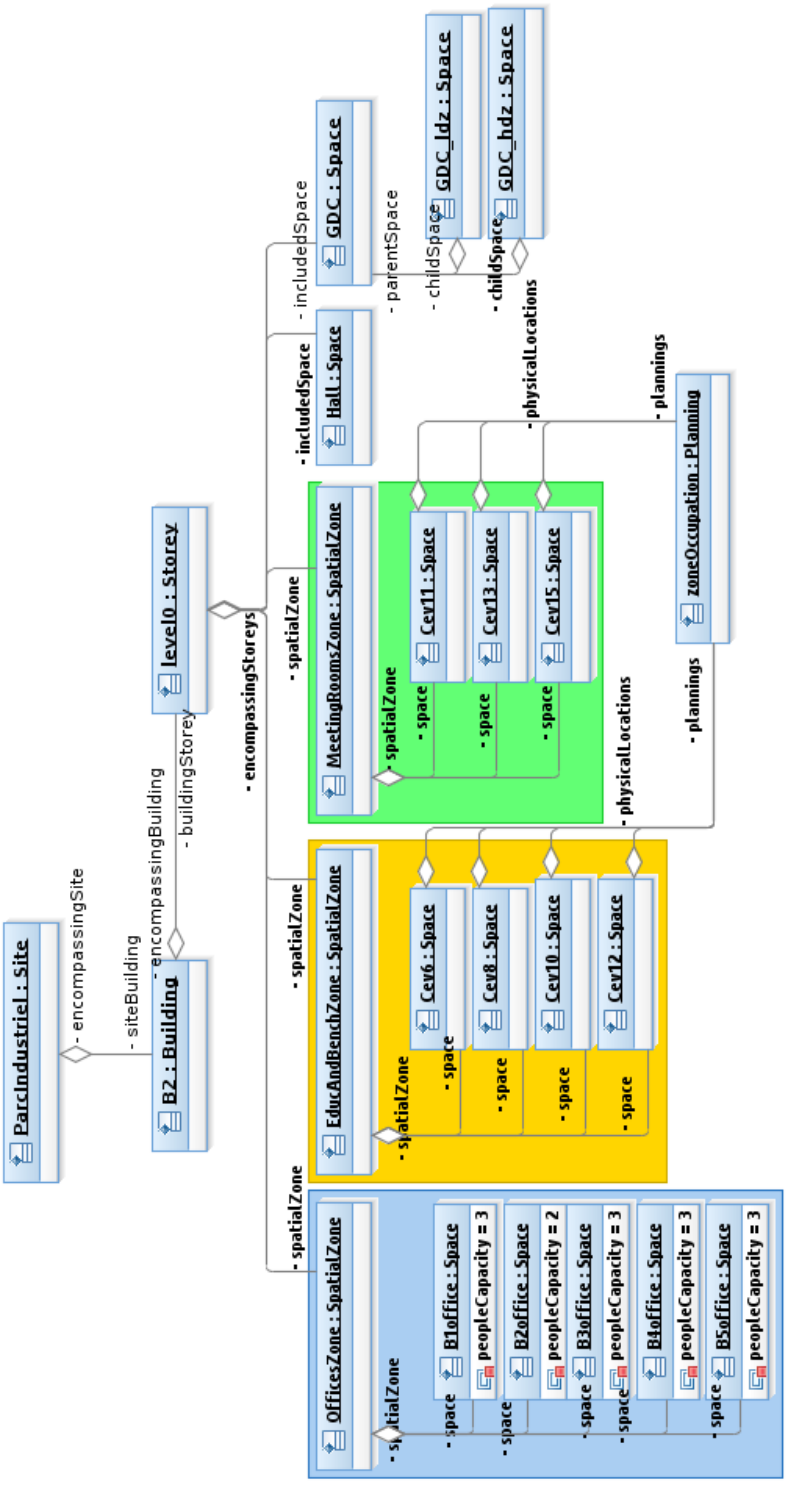

FIGURE 6.15 : Diagramme partiel de la zone optimisée par RIDER

- le système de récupération de l'énergie fatale [3] du green data center,

3. Énergie dégagée, ici, la chaleur des machines, qui serait perdue dans le système de refroidissement à moins d'être réutilisée pour d'autres usages, tels que le chauffage du bâtiment par exemple.

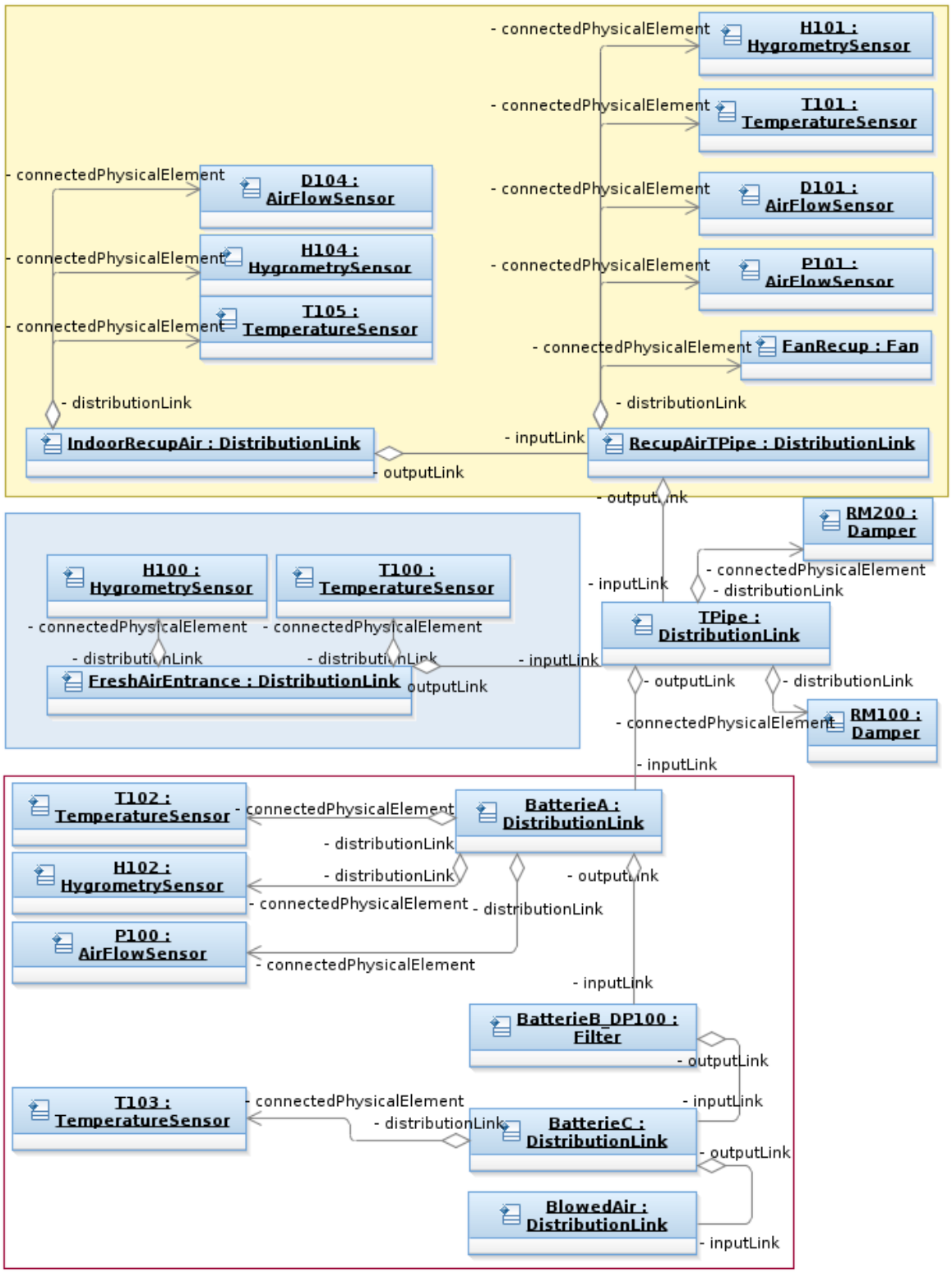

FIGURE 6.16 : Modélisation d'une CTA connectée à RIDER

- les réseaux de distribution d'air neuf et de récupération d'air vicié des locaux,

- les batteries terminales permettant de régler la température et débit d'air soufflé dans chaque bureau.

Les figures 6.17 et 6.18 présentent l'instrumentation du bureau B1 et de la salle de benchmark Cévennes 8. Nous n'avons pas représenté l'ensemble des salles de la zone pilote car elles sont toutes similaires.

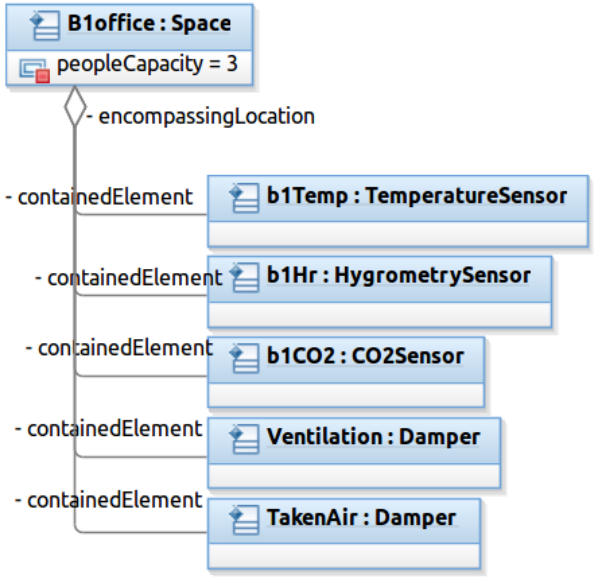

FIGURE 6.17 : Instrumentation du bureau B1

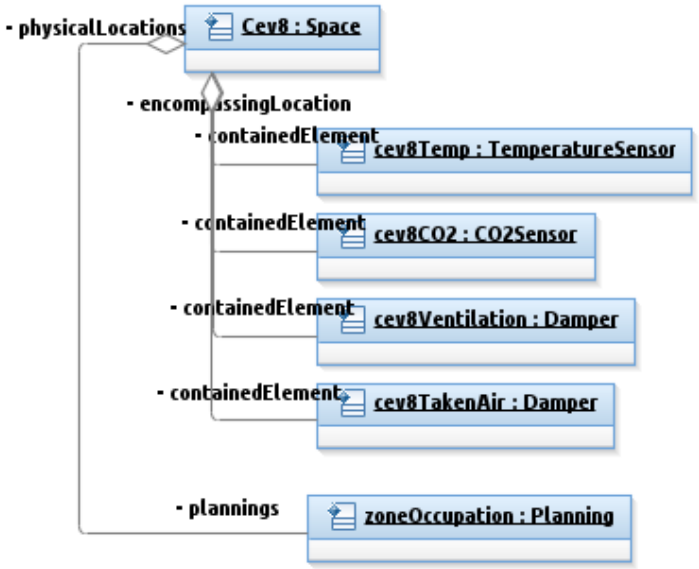

FIGURE 6.18 : Instrumentation de la salle de benchmark cev8

La section suivante décrit le modèle de caractéristiques adapté au bâtiment modélisé par le CMI décrit dans cette section.

6.4 Modèle de caractéristiques adapté à un bâtiment – CSFM

Les diagrammes de caractéristiques présentés dans les figures 6.3, 6.4, 6.5, 6.6, 6.8, 6.9, 6.10, et 6.11 restent identiques dans le CSFM. Les diagrammes présentés dans les figures 6.7 et 6.12 représentent les parties du CM impactées par l'algorithme d'adaptation au contexte.

Les figures 6.20 à 6.23 présentent chacune un sous-ensemble du CSFM contenant les caractéristiques relatives à une instance de concept du contexte. La colonne de gauche présente une instance de concept, et la colonne de droite présente, sous la forme d'une arborescence, les caractéristiques spécifiques à cette instance de concept.

Nous présentons également un exemple de choix de caractéristiques qui forment une configuration permettant d'obtenir un logiciel d'optimisation énergétique similaire au prototype en cours d'installation dans la zone pilote. Les CSF choisies sont représentées en vert.

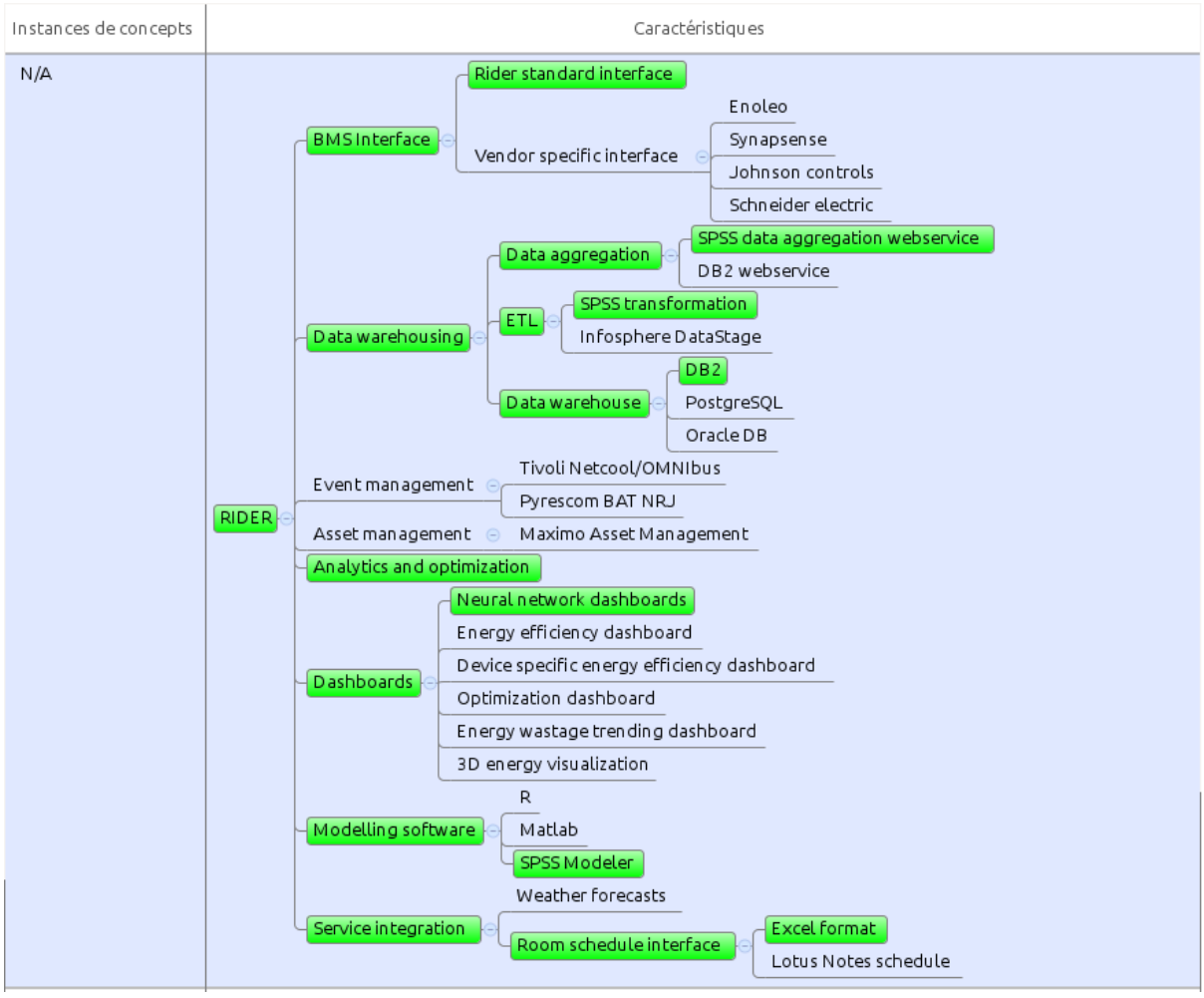

FIGURE 6.19 : Caractéristiques qui ne sont associées à aucune instance de concept

La figure 6.19 présente les caractéristiques spécifiques au contexte (CSF) qui ne sont

associées à aucune instance de concept. Elles sont associées aux caractéristiques du FM, de mêmes noms, qui ne sont associées à aucun concept.

La figure 6.20 présente les CSF associées à l'instance *ParcIndustriel* du concept *Site*. Les CSF présentées dans cette figure sont liées aux caractéristiques montrées dans la figure 6.7 qui sont associées au concept *Site*. La caractéristique racine de ce diagramme (et des suivants) représente "l'ancêtre" le plus proche commun aux caractéristiques dupliquées.

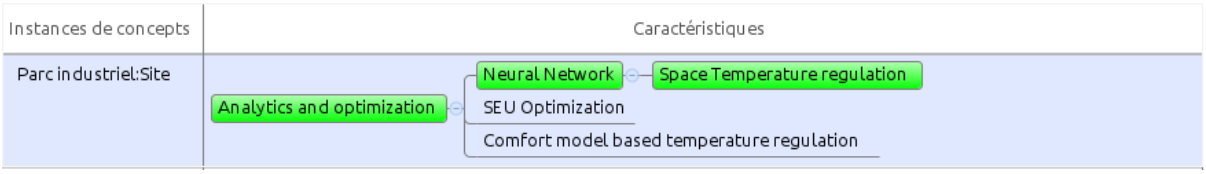

FIGURE 6.20 : Caractéristiques associées à *ParcIndustriel*, instance du concept *Site*

La figure 6.21 montre les CSF associées à l'instance *OfficesZone* du concept *SpatialZone*. Les CSF de cette figure sont liées aux caractéristiques montrées dans la figure 6.13 qui sont associées au concept *SpatialZone*.

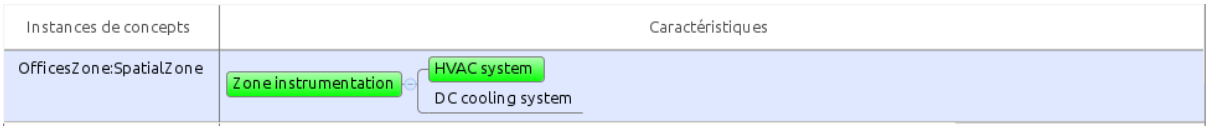

FIGURE 6.21 : Caractéristiques associées à la zone de bureaux *OfficesZone*, instance du concept *SpatialZone*

La figure 6.22 montre les CSF associées à l'instance *B1office* du concept *Space*. Les CSF présentées dans cette figure sont liées aux caractéristiques montrées dans la figure 6.12 qui sont associées au concept *Space*.

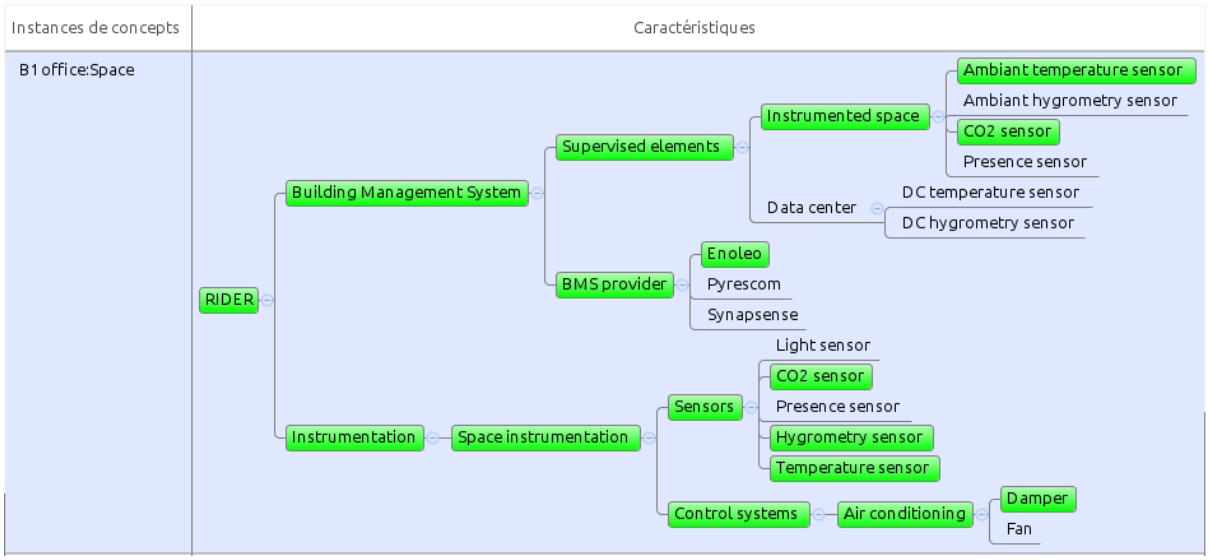

FIGURE 6.22 : Caractéristiques associées au bureau *B1office*, instance du concept *Space*

La figure 6.23 montre les CSF associées à l'instance *Cev8* relative au concept *Space*. Les CSF présentées dans cette figure sont liées aux caractéristiques montrées dans la figure 6.23 qui sont associées au concept *Space*. Les choix effectués diffèrent de ceux faits pour le bureau B1.

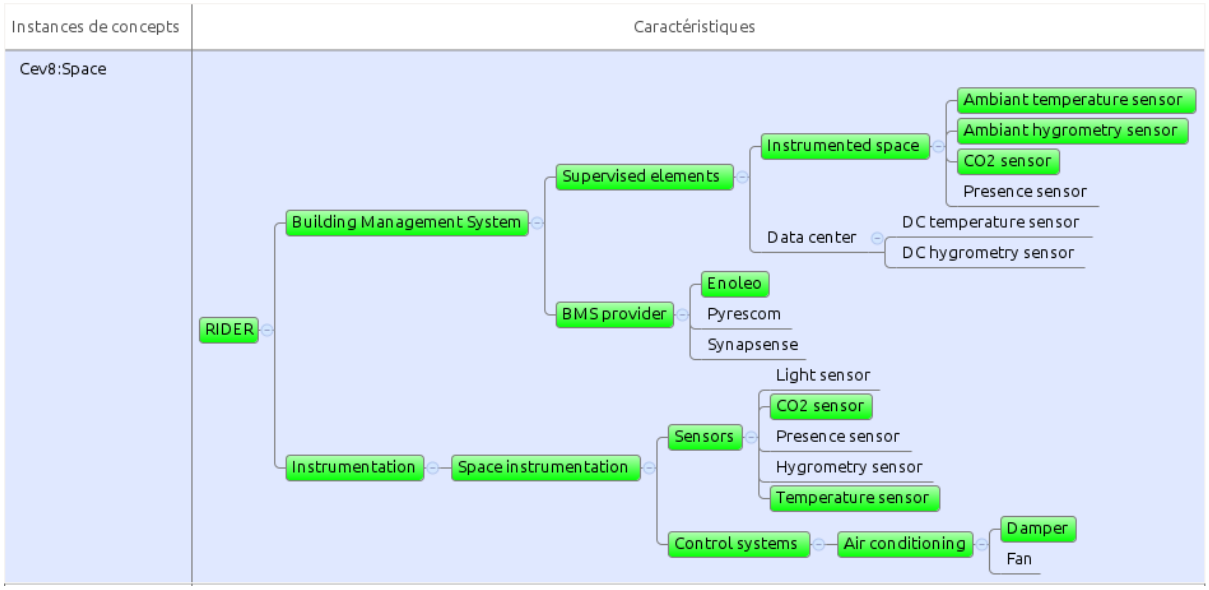

FIGURE 6.23 : Caractéristiques associées à la salle de réunion *Cev8*, instance du concept *Space*

6.5 Conclusion

Nous avons présenté dans ce chapitre un exemple de mise en œuvre de notre approche dans le cadre du prototype du projet RIDER sur la zone pilote du site d'IBM Montpellier. Nous nous sommes appuyés sur le modèle de bâtiments présenté dans le chapitre 2, et sur le méta-modèle de caractéristiques du chapitre 4.

Nous avons tout d'abord proposé un modèle de caractéristiques qui couvre l'ensemble des domaines du projet. Ensuite, nous avons présenté une instance du modèle de bâtiments décrivant la zone pilote. Pour finir, nous avons présenté le résultat de l'algorithme d'adaptation de modèles de caractéristiques au contexte du bâtiment présenté au chapitre 5. La configuration proposée correspond aux choix effectués sur la zone pilote.

L'application au projet RIDER ne permet pas, pour le moment, de bénéficier pleinement des avantages de l'adaptation au contexte d'un modèle de caractéristiques car il y a peu de points de variabilité parmi les caractéristiques de ses logiciels. Nous avons étendu le domaine de la ligne de produits de RIDER aux GTC et à l'instrumentation du bâtiment. Une telle ligne de produits permettrait de configurer dans le détail les GTC en spécifiant quelles zones superviser, et d'adapter en fonction des objectifs d'optimisation l'instrumentation du bâtiment pour avoir le nombre pertinent de capteurs et d'actionneurs.

Le projet est encore jeune. À mesure que de nouvelles installations seront réalisées, des développements ou adaptations spécifiques seront réalisés et donneront lieu à de nouvelles caractéristiques et points de variabilité. Comme le remarque Klaus Pohl *et al.* [2005] l'effort nécessaire pour mettre en œuvre une ligne de produits logiciels commence à être rentable à partir du troisième projet. Lorsque les trois pilotes seront terminés nous pourrons avoir une vision plus claire du modèle de caractéristiques.

Ainsi, cette ligne de produits permet de déterminer si un bâtiment est correctement équipé, et quels sont les équipements manquant pour utiliser une méthode d'optimisation donnée. Comme développement futur, nous pourrions déterminer, en fonction du coût des instruments et des gains énergétiques attendus, quelles sont les méthodes d'optimisation énergétiques les plus adaptées.

La mise en œuvre du CSFM montre l'utilité de vues qui permettraient de représenter, par exemple, l'union des sous-ensembles du CSFM relatifs à un même concept. Cela permettrait de réduire le nombre de CSF à sélectionner lors de la configuration.

CHAPITRE

7 Conclusion et perspectives

Le projet RIDER a pour objectif l'optimisation énergétique de bâtiments. Les logiciels utilisés dans ce domaine pour connecter les instruments aux logiciels de gestion technique centralisée (GTC), et les GTC à des logiciels d'optimisation énergétique, d'entrepôts de données et de tableaux de bord, doivent être configurés dans chaque bâtiment pour déterminer par exemple :

- comment communiquer avec les instruments,
- les zones du bâtiment que doivent contrôler la ou les GTC,
- quelles méthodes d'optimisation utiliser selon les usages du bâtiment (par exemple, la possibilité de faire participer les habitants, la présence de data centers, ...),
- quels tableaux de bord utiliser tout en respectant la confidentialité des mesures effectuées,
- quels logiciels utiliser selon leurs contraintes d'interopérabilité et les fonctionnalités attendues,

Dans le cadre de cette thèse, nous avons travaillé, de façon générale, dans le contexte de la réalisation d'une ligne de produits RIDER. Nous nous sommes intéressés à la configuration d'instances de la ligne de produits RIDER via les modèles de caractéristiques (feature models). Plus précisément, nous avons travaillé sur la proposition d'une méthode permettant l'adaptation du modèle de caractéristiques RIDER au bâtiment à contrôler. La méthodologie que nous proposons n'est cependant pas spécifique à ce projet. Elle peut être réutilisé dans tout domaine nécessitant l'adaptation du modèle de caractéristiques d'une ligne de produits au futur contexte de ses produits.

Afin de déterminer les spécificités de ce type de ligne de produits, et plus particulièrement celles du projet RIDER, nous nous sommes investi avec les autres membres du consortium dans les étapes initiales d'analyse et de conception du projet. Nous avons participé à l'identification des cas d'utilisation, à des interviews avec des experts des différents métiers impliqués, à l'élaboration et l'évaluation d'architectures potentielles, à l'expérimentation de solutions existantes, et à l'analyse et la conception de modèles métier.

Plusieurs visions de l'architecture de RIDER [Xavier Vasques *et al.*, 2011] ont été imaginées successivement :

1. Le contexte (instrumentation, automatisation de bâtiments) et les besoins de RIDER étaient proches de ceux trouvés dans le domaine de l'informatique ubiquitaire. Dans ce domaine, les architectures utilisent parfois un *context broker* (cf. chapitre 2) pour traiter les mesures des capteurs, contrôler les actionneurs et raisonner selon des scénarios pré-établis. Il avait donc été envisagé d'utiliser le moteur de règles JRules afin de coordonner les différents scénarios d'optimisation énergétique et les calculs d'optimisation des modules. Par exemple, les règles auraient pu déterminer, en temps réel, quelle était la température de chauffage ou de climatisation optimale à l'aide d'un indicateur fourni par un module d'analyse statistique.

   Cette première solution nécessitait beaucoup de développement logiciel, mais aurait permis la création une ligne de produits capable de configurer des logiciels avec une granularité très fine, puisque le code aurait pu être généré ou édité pour chaque produit.

2. L'architecture logicielle relative au point précédent pouvait être enrichie par certains logiciels édités par IBM (IBM étant le porteur du projet). Ces logiciels permettent de fournir de très nombreuses fonctionnalités (par exemple, le data warehouse avec *DB2*, le data mining avec *InfoSphere*, les tableaux de bord avec *Cognos*, les analyses statistiques avec *SPSS*, le moteur de règle avec *JRules*, les interfaces avec des systèmes de supervision de bâtiment avec *Maximo for Energy Optimization*, la gestion de la maintenance et de l'état des instruments avec *Maximo Asset Management*). Ces logiciels sont complexes à mettre en œuvre car ils sont destinés à traiter de gros volumes de données dans un contexte industriel.

   Ce second point aurait nécessité la mise en œuvre d'une SPL spécialisée dans le domaine de la configuration de logiciels IBM. Des solutions existent pour automatiser le déploiement de solutions basées sur ces logiciels, cependant, ce type de problématique sort du cadre d'une thèse.

3. La solution IBM *Intelligent Building Management* (IIBM) a été officialisée à la moitié de cette thèse. Elle décrit comment gérer un bâtiment afin, entre autres, d'en optimiser l'énergie. Cette solution réutilise entre autres les logiciels IBM énoncés dans le point précédent et décrit les plans d'architecture selon lesquels les logiciels doivent être mis en œuvre.

   Cette solution peut également grandement bénéficier d'une approche par lignes de produits. Cependant, la mise en œuvre est plus difficile car le paramétrage des logiciels ne peut plus être facilement modifié. De plus, il est difficile de construire un modèle pertinent des caractéristiques d'une telle solution, à moins d'utiliser des outils de rétro-ingénierie capables de bâtir un modèle de caractéristiques à partir de plans d'architectures, de fichiers de configuration et de documentation textuelle. La construction manuelle du modèle de caractéristiques correspondant impose une vision des composants de cette solution en tant que composants de très grosse granularité. La solution en cours de réalisation est prévue pour être intégrée dans cette solution IBM. Elle est basée sur des réseaux de neurones développés dans le logiciel *SPSS*.

Dans le cadre de notre évaluation des différentes solutions envisagées pour RIDER, nous avons également été amené à évaluer les différentes façons de modéliser un bâtiment. Aucun modèle envisagé ne correspondait aux besoins spécifiques à RIDER. Nous donc dû réaliser un modèle permettant de décrire les bâtiments et leurs équipements afin de fournir les informations nécessaires aux algorithmes d'optimisation énergétique de RIDER. Nous avons donc également proposé un nouveau modèle de bâtiment (décrit dans le chapitre 2). Nous l'avons créé en synthétisant différents autres modèles plus spécialisés dans des sous domaines précis (par exemple, les data centers [Ling Tai *et al.*, 2008]) ou très génériques [NBDM, 2008; Gerhard Gröger *et al.*, 2012; gbXML.org, 2013]. Nous y avons aussi ajouté de nouveaux concepts issus des besoins identifiés lors des entretiens avec les membres du consortium spécialisés dans la gestion ou l'optimisation énergétique de bâtiments.

Ce travail sur ces logiciels, architectures et modèles de bâtiments [Xavier Vasques *et al.*, 2011] a été long mais nous a permis d'avoir une compréhension globale de l'ensemble du projet RIDER et des caractéristiques du domaine de l'optimisation énergétique [Thibaut Possompès *et al.*, 2010a].

Nous avons ensuite proposé, dans le chapitre 4, une synthèse des méta-modèles de caractéristiques existants [Thibaut Possompès *et al.*, 2010b]. En effet, il existe de nombreuses contributions améliorant chacune des points spécifiques, mais il nous manquait un méta-modèle contenant l'ensemble des améliorations pertinentes pour la mise en œuvre de la ligne de produits RIDER. Nous avons également inclus une nouvelle façon d'associer caractéristiques et concepts du contexte [Thibaut Possompès *et al.*, 2011a]. C'est à dire, une façon de montrer la dépendance entre une caractéristique du logiciel et un élément du contexte. Par exemple, une caractéristique du logiciel *Mesure de la température* nécessite la présence de capteurs de température dans le bâtiment.

Nous avons ensuite proposé une méthodologie, dans le chapitre 5, permettant d'adapter un modèle de caractéristiques au contexte dans lequel doit être déployé pour s'exécuter, un nouveau produit [Thibaut Possompès *et al.*, 2011b, 2013]. Cette méthode permet de garantir que toutes les caractéristiques proposées à la configuration y seront utilisables.

Le contexte de RIDER ne nous a pas permis de montrer toutes les possibilités de notre approche car la majorité des composants logiciels ne sont pas décomposables (logiciels propriétaires) et de grosse granularité. Cela limite grandement le nombre de choix possible parmi les caractéristiques de ces composants. Cependant, notre approche est générique et peut être appliquée à n'importe quel domaine.

Perspectives

Aujourd'hui, notre approche est capable de suggérer une ou plusieurs solutions (à travers l'ensemble des configurations possibles d'un CSFM). Cependant, nous pouvons étendre notre approche pour raffiner les choix possibles en se basant sur certains critères tels que :

- le choix d'un maximum ou minimum de caractéristiques, cela permettrait de quantifier la compatibilité de l'instrumentation d'un bâtiment avec la ligne de produits,
- la solution avec le moins de coûts de licences logicielles, ce critère permettrait de chiffrer plus précisément le coût du déploiement d'un produit RIDER.
- la solution qui maximise le nombre d'éléments du contexte utilisés afin de tirer profit au mieux des instruments du bâtiment,
- l'utilisation des méthodes d'optimisation les plus adaptées au bâtiment. Ce critère serait utile pour choisir parmi plusieurs méthodes d'optimisation dont l'efficacité diffère en fonction de la nature du bâtiment ciblé.

La création de vues sur le CSFM permettrait d'en simplifier la configuration. Ces vues nécessiteraient un outil capable d'effectuer les actions suivantes :

- Visualiser le CSFM graphiquement et filtrer les caractéristiques du CSFM pour n'afficher que celles pertinentes à certaines parties prenantes. Cette vue permettrait de ne pas surcharger l'interface de configuration avec des caractéristiques non pertinentes pour les personnes qui les choisissent.
- Manipuler et interroger à l'aide de requêtes le CSFM et le CMI afin de, par exemple, afficher les parties du CSFM relatives à des éléments du contexte satisfaisant certains critères (les pièces dont la surface est supérieure à 30 m²). Cette vue permettrait de faciliter la navigation dans les éléments du contexte car ce modèle peut être complexe et constitué d'un très grand nombre d'éléments.
- Afficher, sous la forme d'un même sous arbre du CSFM, les caractéristiques qui ont été dupliquées (*i.e.*, les CSF associées à une même GF) et permettre leur configuration simultanément. Cette vue permettrait de choisir simultanément un ensemble de caractéristiques du CSFM et permettrait d'éviter une sélection manuelle et laborieuse des caractéristiques spécifiques au contexte dupliquées à partir de la même caractéristique.
- Visualiser, pour une caractéristique associée à des CSF, quels sont les éléments du contexte auxquels les CSF correspondantes sont, ou peuvent être, associées. Cette vue serait utile pour obtenir une vision d'ensemble des zones associées à des caractéristiques.
- Visualiser les éléments du contexte qui ne sont pas associés à des CSF car ils ne remplissaient pas les conditions requises. Dans le contexte de RIDER, cette vue serait utile pour voir quelles zone d'un bâtiment ne seront pas contrôlées par le futur produit RIDER.

Nous pourrions permettre la configuration de nouveaux produits en fonction des caractéristiques de produits existants dans le but de les faire communiquer et collaborer. Par exemple, dans le cadre d'un logiciel d'optimisation énergétique multi-échelles, la configuration d'un nouveau produit pourrait être réalisée en fonction des interfaces de communication des produits existants.

Notre méthode d'adaptation au contexte pourrait être réutilisée pour réaliser cela. Le modèle de contexte (CM) étant dans ce cas un modèle de caractéristiques (FM), et l'instance du modèle de contexte (CMI) étant l'ensemble des configurations des produits existants. Ainsi, une caractéristique pourrait être associée par le lien *ModelRelationship* à d'autres

caractéristiques. Une caractéristique spécifique au contexte (CSF) serait alors créée pour chaque caractéristique présente dans les produits existants.

Par ailleurs, nous pourrions automatiser l'extraction des contraintes OCL du modèle de caractéristiques (FM) dans le but de vérifier qu'une configuration du modèle de caractéristiques adapté au contexte (CSFM) est valide.

Un autre développement intéressant serait d'intégrer à Rational Software Architect (RSA) la génération du CSFM. Cela permettrait de totalement intégrer l'approche dans un même logiciel.

La gestion des assets pourrait être implémentée également dans RSA afin de bénéficier des outils intégrés pour la gestion d'un dépôt d'assets. L'intérêt d'une telle intégration dans un contexte industriel serait de faciliter l'utilisation des logiciels IBM permettant l'automatisation du déploiement des assets constituant un nouveau produit. De plus, comme nous l'avons vu dans le cas du projet RIDER, de nombreux partenaires sont amenés à collaborer et à contribuer aux développement des nouveau logiciels. La plateforme collaborative IBM Jazz est intégrée à RSA. Cela nous permettrait d'apporter à nos outils la possibilité de les utiliser de façon collaborative. Les différentes parties prenantes auraient ainsi la possibilité de décrire les caractéristiques de leurs développements respectifs.

Un autre développement possible consisterait à réaliser une rétro-ingénierie des assets stockés dans un dépôt RAS. Cette norme contient la possibilité de spécifier des points de variabilité afin de décrire les différentes possibilités de les combiner ou de les adapter aux besoins liés à leur réutilisation.

ANNEXE

A Profil UML

A.1 Présentation EBNF du nouveau méta-modèle de caractéristiques

Cette section décrit la grammaire, créée à partir de notre méta-modèle de caractéristiques, permettant de définir un modèle de caractéristiques textuellement. Nous présentons d'abord la grammaire textuellement sous la forme EBNF, puis graphiquement avec des diagrammes syntaxiques.

A.1.1 Forme EBNF

```
AssetArtifact ::= Name BindingTime AssetRepository

AssetRepository ::= Name URL (AssetArtifact)*

BindingTime ::= "Runtime" | "Deployment time"

Class ::= [http://slps.github.io/zoo/uml/uml2.html#Class]

Concern ::= Name (Stakeholder | FeatureSet | Layer)*

Constraint ::= [http://slps.github.io/tank/ocl/expressions.bnf]
```

```
Digits    ::= [0—9]+

DirectedBinaryRelationship ::= Multiplicity (Detail | Enrich | Implement) "parent"
    Feature "child" Feature

Feature ::= Name Multiplicity ("parent" DirectedBinaryRelationship ("groupedBy"
    RelationshipGroup)?)? (AssetArtifact |"child" DirectedBinaryRelationship | "
    conflicts" Feature | ModelElement | Property | "recommends" Feature | "requires"
    Feature | RelationshipGroup)*

FeatureOccurrence ::= Feature ModelElementInstance?

FeatureSet ::= Name (FeatureSet | Feature)*

Layer ::= Name (Layer | FeatureSet | Feature)*

ModelElement ::= Name ModelElementReference ModelElementSelectionRule?

ModelElementInstance ::= Name ModelElementReference

ModelElementReference ::= Name Class (ModelElementInstance)* (ModelElement)*

ModelElementSelectionRule ::= Constraint+

Modification ::= Constraint+ "source" Property "cible" Property

Multiplicity ::= "(" Digits ", " Digits ")"

Name ::= [A—Z] String

Product ::= Name (FeatureOccurrence)*

ProductLine ::= (Feature | FeatureSet | Layer)+

Property ::= Name Value VariabilityType Modification*

RelationshipGroup ::=  ( Multiplicity | "Or" | "And" | "Xor") "root" Feature "groups"
     Feature

Stakeholder ::= Name (Concern)*

String ::= [a—zA—Z0—9]+

URL ::= 'http:/' (String '/'?)+

Value ::= String | Digits

VariabilityType ::= "Fixed" | "Variable" | "Family variable" | "User defined"
```

A.1.2 Diagrammes syntaxiques

FIGURE A.1 : AssetArtifact

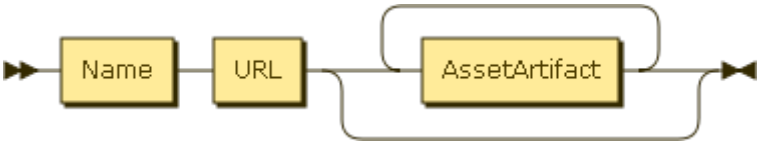

FIGURE A.2 : AssetRepository

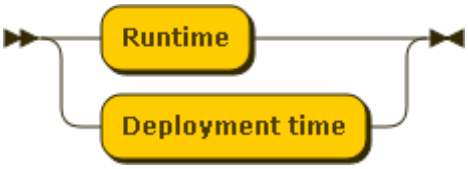

FIGURE A.3 : BindingTime

FIGURE A.4 : Class

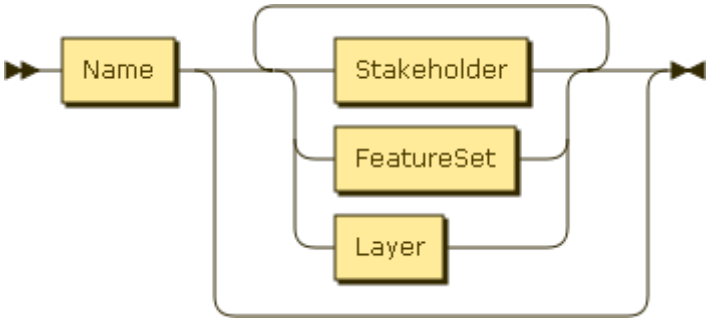

FIGURE A.5 : Concern

FIGURE A.6 : Constraint

FIGURE A.7 : Digits

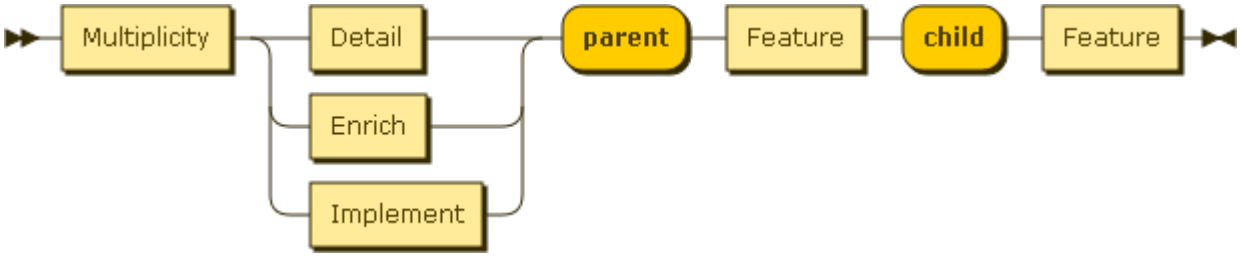

FIGURE A.8 : DirectedBinaryRelationship

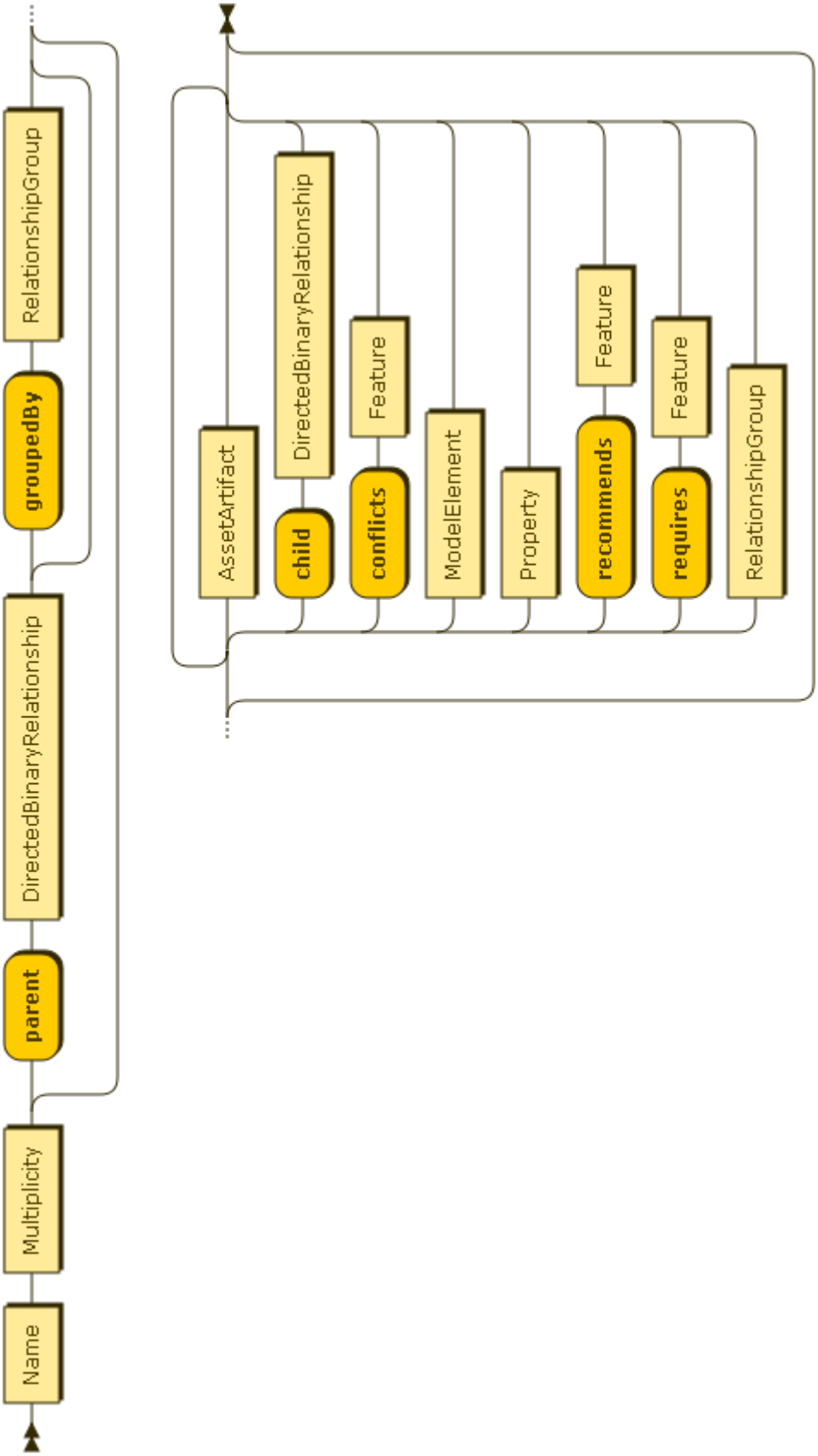

FIGURE A.9 : Feature

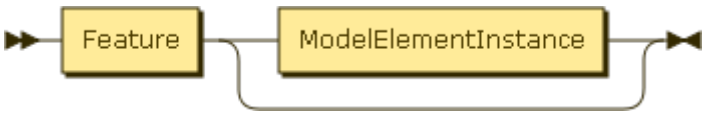

FIGURE A.10 : FeatureOccurrence

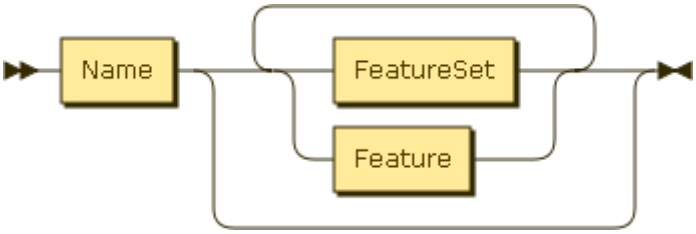

FIGURE A.11 : FeatureSet

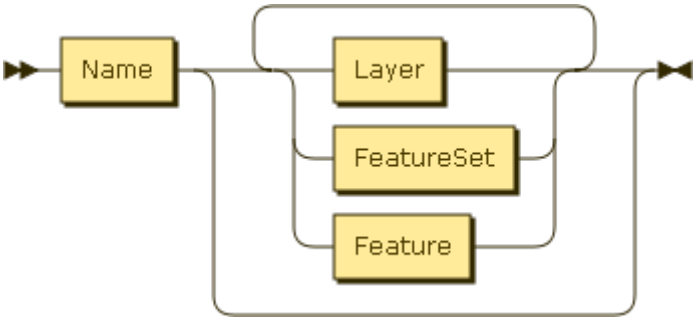

FIGURE A.12 : Layer

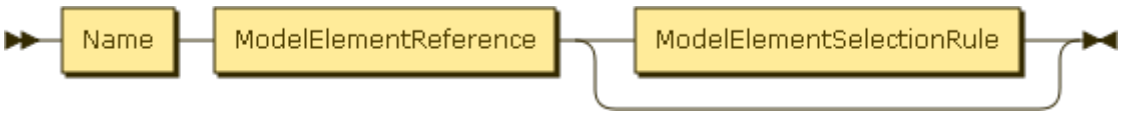

FIGURE A.13 : ModelElement

FIGURE A.14 : ModelElementInstance

FIGURE A.15 : ModelElementReference

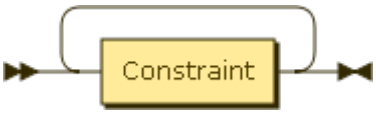

FIGURE A.16 : ModelElementSelectionRule

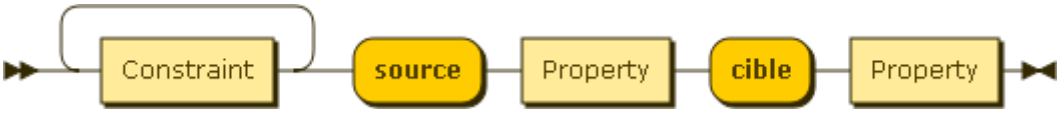

FIGURE A.17 : Modification

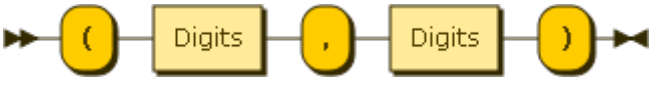

FIGURE A.18 : Multiplicity

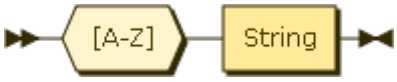

FIGURE A.19 : Name

FIGURE A.20 : Product

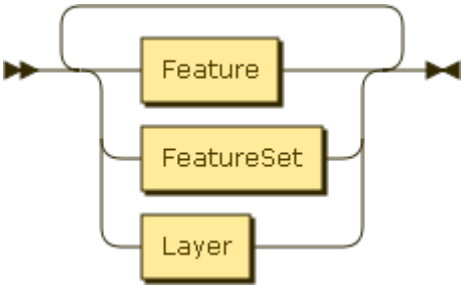

FIGURE A.21 : ProductLine

FIGURE A.22 : Property

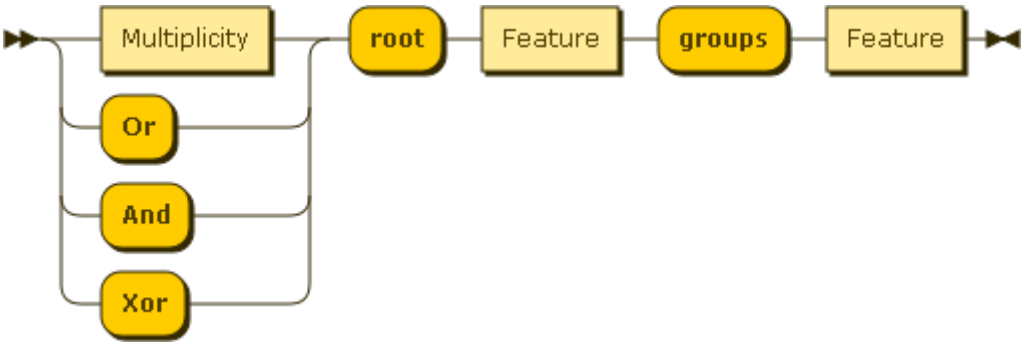

FIGURE A.23 : RelationshipGroup

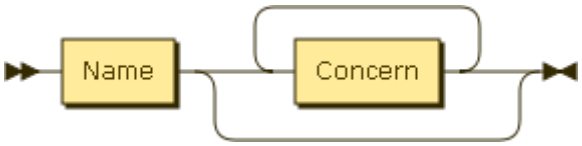

FIGURE A.24 : Stakeholder

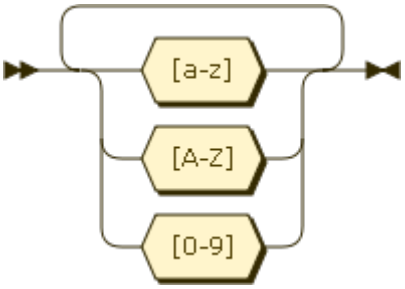

FIGURE A.25 : String

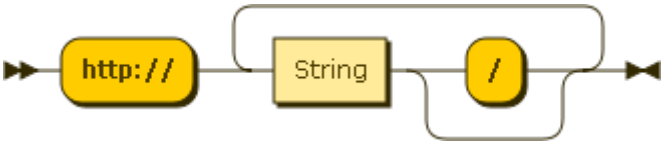

FIGURE A.26 : URL

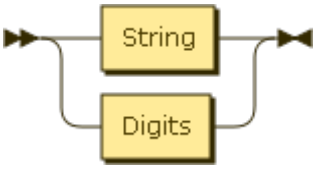

FIGURE A.27 : Value

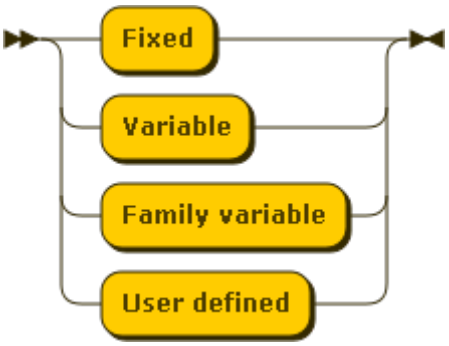

FIGURE A.28 : VariabilityType

Bibliographie

MohdSyazwan Abdullah, Chris Kimble, Richard Paige, Ian Benest et Andy Evans : Developing a UML profile for modelling knowledge-based systems. *In* Uwe Aßmann, Mehmet Aksit et Arend Rensink, éditeurs : *Model Driven Architecture*, volume 3599 de *Lecture Notes in Computer Science*, pages 220–233. Springer Berlin Heidelberg, janvier 2005. ISBN 978-3-540-28240-2. URL http://dx.doi.org/10.1007/11538097_15. Cité page 18.

Gregory Abowd, Anind Dey, Peter Brown, Nigel Davies, Mark Smith et Pete Steggles : Towards a better understanding of context and context-awareness. *In* Hans-W. Gellersen, éditeur : *Handheld and Ubiquitous Computing*, volume 1707 de *Lecture Notes in Computer Science*, pages 304–307. Springer Berlin / Heidelberg, 1999. ISBN 978-3-540-66550-2. URL http://www.springerlink.com.gate6.inist.fr/content/pwpmm42n3krr1f3a/abstract/. Cité page 12.

M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan et J. P Rigault : Modeling context and dynamic adaptations with feature models. *In 4th International Workshop Models@ run. time at Models*, volume 9, 2009. Cité pages 2, 34, 35, 36 et 37.

M. Acher, P. Collet, P. Lahire et R. B France : A domain-specific language for managing feature models. *In Proceedings of the 2011 ACM Symposium on Applied Computing*, page 1333–1340, 2011a. Cité pages 34 et 35.

Mathieu Acher, Philippe Collet, Philippe Lahire et Robert France : Composing feature models. *In* Mark van den Brand, Dragan Gaševic et Jeff Gray, éditeurs : *Software Language Engineering*, volume 5969 de *Lecture Notes in Computer Science*, pages 62–81. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-12106-7. URL http://www.springerlink.com.gate6.inist.fr/content/rt7546169p4174v8/abstract/. Cité page 36.

Mathieu Acher, Philippe Collet, Philippe Lahire et Robert B. France : Managing feature models with familiar : a demonstration of the language and its tool support. *In Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, page 91–96, 2011b. URL http://dl.acm.org/citation.cfm?id=1944903. Cited by 0004. Cité page 65.

H. Akbari : Shade trees reduce building energy use and CO2 emissions from power plants. *Environmental Pollution*, 116:S119–S126, 2002. Cité page 8.

Ali Keçebaş et İsmail Yabanova : Thermal monitoring and optimization of geothermal district heating systems using artificial neural network : A case study. *Energy and Buildings*, 2010. ISSN 0378-7788. URL http://www.sciencedirect.com/science/article/pii/S0378778812002046?v=s5. Cité page 11.

Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba et Carlos Lucena : Refactoring product lines. *In Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, page 201–210, New York, NY, USA, 2006. ACM. ISBN 1-59593-237-2. URL http://doi.acm.org.gate6.inist.fr/10.1145/1173706.1173737. Cité page 36.

António Grilo et Ricardo Jardim-Goncalves : Challenging electronic procurement in the AEC sector : A BIM-based integrated perspective. *Automation in Construction*, 20(2):107–114, mars 2011. ISSN 0926-5805. URL http://www.sciencedirect.com/science/article/pii/S0926580510001378. Cité page 16.

T. Asikainen, T. Mannisto et T. Soininen : A unified conceptual foundation for feature modelling. *In SPLC '06 : Proceedings of the 10th International on Software Product Line Conference*, page 31–40. IEEE Computer Society, 2006. ISBN 0-7695-2599-7. Cité page 33.

Felix Bachmann et Len Bass : Managing variability in software architectures. *ACM SIGSOFT Software Engineering Notes*, 26(3):126–132, 2001. URL http://dl.acm.org/citation.cfm?id=375274. Cité page 29.

M. Baldauf, S. Dustdar et F. Rosenberg : A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007. Cité page 12.

Don Batory : Feature models, grammars, and propositional formulas. *In* David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Henk Obbink et Klaus Pohl, éditeurs : *Software Product Lines*, volume 3714, pages 7–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-28936-4, 978-3-540-32064-7. URL http://www.springerlink.com.gate6.inist.fr/content/9t1veyhqe6myj08r/. Cité page 33.

David Benavides, Sergio Segura et Antonio Ruiz-Cortés : Automated analysis of feature models 20 years later : A literature review. *Information Systems*, 35(6):615–636, septembre 2010. ISSN 0306-4379. URL http://www.sciencedirect.com/science/article/pii/S0306437910000025. Cité page 33.

Benoit Lange : *Visualisation interactive de données hétérogènes pour l'amélioration des dépenses énergétiques du bâtiment.* Thèse de doctorat, Université Montpellier II, 2012. Cité page 8.

Y. Bontemps, P. Heymans, P. Y Schobbens et J. C Trigaux : Semantics of FODA feature diagrams. *In Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation–Towards Tool Support*, page 48–58, 2004. Cité pages 28, 42 et 87.

Conrad Boton, Sylvain Kubicki et Gilles Halin : Adaptation of user views to business requirements : towards adaptive views models. *In Conference Internationale Francophone sur l'Interaction Homme-Machine*, pages 113–116, Luxembourg, Luxembourg, 2010. ACM. ISBN 978-1-4503-0410-8. Cité page 30.

Brad Brech, Ravirajan Rajan, James Fletcher, Colin Harrison, Michael Hayes, John Hogan, Lisa Hopkins, Pamela K. Isom, John Meegan, Claire Penny, Jane L. Snowdon et Doug A. Wood : IBM redbooks | smarter cities series : Understanding the IBM approach to efficient buildings, 2011. URL http://www.redbooks.ibm.com/abstracts/redp4735.html. Cité pages 1, 11, 12, 15 et 100.

buildingSMART : Industry foundation classes, 2011. URL http://buildingsmart-tech.org/specifications/ifc-overview/ifc-overview-summary. Cité page 16.

buildingSMART : Industry foundation classes (IFC) overview summary, 2013a. URL http://www.buildingsmart-tech.org/specifications/ifc-overview/ifc-overview-summary. Cité page 16.

buildingSMART : List of all software applications/utilities in the IFC-Compatible implementations database., 2013b. URL http://www.buildingsmart-tech.org/implementation/implementations. Cité page 16.

buildingSMART : Scope of the IFC specification, 2013c. URL http://www.buildingsmart-tech.org/ifc/IFC4/final/html/schema/chapter-1.htm. Cité page 16.

Günter Böckle, Klaus Pohl et Frank Linden : A framework for software product line engineering. *In Software Product Line Engineering*, pages 19–38. Springer-Verlag, Berlin/Heidelberg, 2005. ISBN 3-540-24372-0. URL http://www.springerlink.com.gate6.inist.fr/content/mk2k7q4506756335/. Cité page 28.

P. C Clements, L. G Jones, L. M Northrop et J. D McGregor : Project management in a software product line organization. *Software, IEEE*, 22(5):54–62, 2005. Cité pages 28 et 29.

K. Czarnecki, S. Helsen et U. Eisenecker : Formalizing cardinality-based feature models and their staged configuration. *University of Waterloo*, 2004a. Cité pages 36, 41, 42, 44, 47 et 77.

K. Czarnecki, S. Helsen et U. Eisenecker : Staged configuration using feature models. *In Lecture notes in computer science*, volume 3154, page 266–283, 2004b. URL http://www.springerlink.com.gate6.inist.fr/content/n3wn2b2dmumke6lu/abstract/. Cité pages 36, 44 et 78.

K. Czarnecki, S. Helsen et U. Eisenecker : Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10(1):7–29, 2005. Cité pages 33, 34, 35, 41, 42, 44, 46 et 47.

Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid et Andrzej W\kasowski : Cool features and tough decisions : a comparison of variability modeling approaches. *In Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, page 173–182, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1058-1. URL http://doi.acm.org.gate6.inist.fr/10.1145/2110147.2110167. Cité page 35.

Daniel Kullmann, Henrik Bindner, I. Cirac et P. Zoller : Elements enabling high-level communication in power systems, 2010. URL http://www.4thintegrationconference.com/posters.asp. Cité page 11.

Daniel Kullmann et Henrik W. Bindner : Using rules in high-level communication for the control of power systems. *In Proceedings of the 2nd International Conference on Microgeneration and Related Technologies*, 2011. URL http://microgen11.supergen-hidef.org/microgenII/CD/full_papers/p168vFINAL.pdf. Cité pages 9 et 11.

A.I. Dounis et C. Caraiscos : Advanced control systems engineering for energy and comfort management in a building environment—A review. *Renewable and Sustainable Energy Reviews*, 13(6–7):1246–1261, août 2009. ISSN 1364-0321. URL http://www.sciencedirect.com/science/article/pii/S1364032108001457. Cité page 8.

W. Keith Edwards et Rebecca E. Grinter : At home with ubiquitous computing : Seven challenges. page 256–272. Springer-Verlag, 2001. Cité page 12.

P. Fernandes, C. Werner et L. Murta : Feature modeling for context-aware software product lines. *In Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering (San Francisco, CA, USA, 2008)*, 2008. Cité pages 34, 36 et 37.

P. Fernandes, C. Werner et E. Teixeira : An approach for feature modeling of context-aware software product line. *Journal of Universal Computer Science*, 17(5):807–829, 2011. URL http://www.jucs.org/jucs_17_5/an_approach_for_feature. Cité pages 2, 34 et 37.

D. Fey, R. Fajta et A. Boros : Feature modeling : A meta-model to enhance usability and usefulness. *In Software Product Lines*, volume 2379 de *Lecture Notes in Computer Science*, page 198–216. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-43985-1. URL http://www.springerlink.com/content/hf8kfqv3c0ly4575. Cité pages 2, 33, 34, 35, 41, 42, 43, 44, 45, 46, 47 et 48.

Franck Fleurey, Oystein Haugen, Birger Moller-Pedersen, Goran K. Olsen, Andreas Svendsen et Xiaorui Zhang : A generic language and tool for variability modeling. Rapport technique SINTEF A13505, SINTEF, Oslo, Norway, 2009. URL http://modelingwizards.isti.cnr.it/wp-content/uploads/2010/10/SINTEF-A13505-Report.pdf. Cité page 28.

gbXML.org : Green building XML schema, 2013. URL http://www.gbxml.org/. Cité pages 16 et 121.

Gerhard Gröger, homas H. Kolbe, Claus Nagel et Karl-Heinz Häfele : OGC city geography markup language (CityGML) encoding standard. Rapport technique, 2012. Cité pages 17 et 121.

V. Gholipour, J.C. Bignon et L. Morel-Guimaraes : Les eco-modèles. un outil pour la conception d'édifices durables. 2009. URL http://www.crai.archi.fr/ninter-dev/detail_publi.php?publi=517. Cité pages 8 et 15.

Hassan Gomaa et Michael E. Shin : Automated software product line engineering and product derivation. *In System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, page 285a, janvier 2007. Cité page 65.

Iris Groher et Markus Voelter : XWeave : models and aspects in concert. *In Proceedings of the 10th international workshop on Aspect-oriented modeling*, AOM '07, page 35–40, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-658-5. URL http://doi.acm.org.gate6.inist.fr/10.1145/1229375.1229381. Cité page 29.

Günter Halmans et Klaus Pohl : Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2(1):15–36, mars 2003. ISSN 1619-1366, 1619-1374. URL http://www.springerlink.com.gate6.inist.fr/content/8r9fflq08gclme25/. Cité pages 30, 31, 33, 34 et 35.

G. Halin et S. Kubicki : Architecture dirigée par les modèles pour une représentation multi-vues du contexte de coopération. *In Proceedings of the 17th international conference on Francophone sur l'Interaction Homme-Machine*, page 211–214, 2005. Cité page 16.

G. Halin et S. Kubicki : Une approche par les modèles pour le suivi de l'activité de construction d'un bâtiment. 2007. Cité page 16.

Harry Chen et Anupam Joshi : Semantic web in the context broker architecture. *In Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. PerCom 2004.*, pages 277—286, 2004. ISBN 0-7695-2090-1. Cité page 12.

Hassan Gomaa et Michael E. Shin : Variability modeling in model-driven software product line engineering. *In Proceedings of the 2nd International Workshop on Model-driven Product Line Engineering (MDPLE 2010)*, 2010. URL http://ceur-ws.org/Vol-625/MDPLE2010-paper3-Gomaa.pdf. Cité page 65.

T. Hofer, W. Schwinger, M. Pichler, G. Leonhartsberger, J. Altmann et W. Retschitzegger : Context-awareness on mobile devices - the hydrogen approach. *In System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, page 10 pp., janvier 2003. Cité page 12.

W. Huang et H.N. Lam : Using genetic algorithms to optimize controller parameters for HVAC systems. *Energy and Buildings*, 26(3):277–282, 1997. ISSN 0378-7788. URL http://www.sciencedirect.com/science/article/pii/S037877889700008X. Cité page 9.

ISO/IEC/IEEE : Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765 :2010(E)*, pages 1 –418, 2010. Cité page 28.

Paul Istoan, Jacques Klein, Gilles Perrouin et Jean-Marc Jézéquel : Survey and classification of software product line variability modelling techniques. 2011. Cité page 28.

Z. Jaroucheh, Xiaodong Liu et S. Smith : A perspective on middleware-oriented context-aware pervasive systems. *In Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 2, pages 249 –254, juillet 2009. Cité page 12.

Z. Jaroucheh, Xiaodong Liu et S. Smith : CANDEL : product line based dynamic context management for pervasive applications. *In 2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 209–216. IEEE, février 2010a. ISBN 978-1-4244-5917-9. Cité pages 34 et 37.

Z. Jaroucheh, Xiaodong Liu et S. Smith : Mapping features to context information : Supporting context variability for context-aware pervasive applications. *In Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*, volume 1, pages 611 –614, 2010b. Cité pages 34 et 37.

Jim Steel, Robin Drogemuller et Bianca Toth : Model interoperability in building information modelling. *Software and Systems Modeling (SoSyM)*, pages 1–11, 2010. ISSN 1619-1366. URL http://dx.doi.org/10.1007/s10270-010-0178-4. 10.1007/s10270-010-0178-4. Cité page 16.

Soteris A. Kalogirou : Applications of artificial neural-networks for energy systems. *Applied Energy*, 67(1–2):17–35, septembre 2000. ISSN 0306-2619. URL http://www.sciencedirect.com/science/article/pii/S0306261900000052. Cité page 9.

Soteris A. Kalogirou : Artificial neural networks in renewable energy systems applications : a review. *Renewable and Sustainable Energy Reviews*, 5(4):373–401, décembre 2001. ISSN 1364-0321. URL http://www.sciencedirect.com/science/article/pii/S1364032101000065. Cité page 9.

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak et A. S. Peterson : Feature-oriented domain analysis (FODA) feasibility study. Rapport technique, Carnegie-Mellon University Software Engineering Institute, novembre 1990. Cité pages 2, 28, 33, 34, 35, 41, 42 et 45.

K. C Kang, S. Kim, J. Lee, K. Kim, E. Shin et M. Huh : FORM : a feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998. Cité pages 2, 34, 35, 41 et 43.

K. C Kang, J. Lee et P. Donohoe : Feature-oriented product line engineering. *IEEE software*, page 58–65, 2002. Cité pages 32, 42 et 43.

S.L. Kiani, M. Knappmeyery, N. Baker et B. Moltchanov : A federated broker architecture for large scale context dissemination. *In Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2964 –2969, juillet 2010. Cité page 12.

Klaus Pohl, Gunter Bockle et Frank van der Linden : *Software product line engineering : Foundations, Principles, and Techniques*, volume 10. Springer, 2005. URL http://cs5292.userapi.com/u11728334/docs/7ca820593ad0/Klaus_Pohl_Software_Product_Line_Engineering.pdf. Cité pages 2, 28, 29 et 118.

D. Kolokotsa, K. Kalaitzakis, E. Antonidakis et G. S. Stavrakakis : Interconnecting smart card system with PLC controller in a local operating network to form a distributed energy management and control system for buildings. *Energy conversion and Management*, 43(1): 119–134, 2002. URL http://www.sciencedirect.com/science/article/pii/S0196890401000139. Cité pages 1 et 8.

S. Kubicki, J.C. Bignon et G. Halin : Building construction coordination by an adaptive representation of the cooperation context. *In Joint International Conference on Computing and Decision Making in Civil and Building Engineering*, 2006. Cité page 16.

Kwang-Soon Choi, Eunseok Choi et Ha-Bong Chung : Design and implementation of energy saving system. *In Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, page 100–103, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1087-1. URL http://doi.acm.org.gate6.inist.fr/10.1145/2103380.2103400. Cité page 10.

Julien Lancia : *Infrastructure orientée service pour le développement d'application ubiquitaires*. Thèse de doctorat, Université de Bordeaux 1, 2008. Cité page 12.

S. Lee, J. Chang et S. Lee : Survey and trend analysis of context-aware systems. *Information-An International Interdisciplinary Journal*, 14(2):527–548, 2011. Cité page 12.

Linda M. Northrop et Paul C. Clements : *A Framework for Software Product Line Practice, Version 5.0*. Carnegie Mellon, Software Engeneering Institute, 2007. URL http://www.sei.cmu.edu/productlines/frame_report/. Cité pages 28, 29 et 32.

Ling Tai, Ron Baker, Elizabeth Edmiston et Ben Jeffcoat : IBM redbooks | IBM tivoli common data model : Guide to best practices. Red paper, novembre 2008. URL http://www.redbooks.ibm.com/abstracts/redp4389.html. Cité pages 17 et 121.

D. Livengood et R. C. Larson : The energy box : Locally automated optimal control of residential electricity usage. *Service Science*, 1(1):1–16, 2009. Cité page 9.

Manuel Bauer : *Gestion biomimétique de l'énergie dans le bâtiment*. Thèse de doctorat, EPFL, Lausanne, 1998. Cité page 9.

Matthias Clauß : *Untersuchung der Modellierung von Variabilität in UML*. Technische Universit\ät Dresden, Diplomarbeit, 2001. Cité page 65.

Mohamad El Achkar : Heuristiques pour la gestion d'énergie dans les bâtiments intelligents, 2010. URL http://www.spectrosciences.com/spip.php?article122. Cité page 11.

B. Morin, F. Fleurey, N. Bencomo, J. M Jézéquel, A. Solberg, V. Dehlen et G. Blair : An aspect-oriented and model-driven approach for managing dynamic variability. *Model Driven Engineering Languages and Systems*, page 782–796, 2010. Cité page 36.

J. Munoz, V. Pelechano et C. Cetina : Implementing a pervasive meetings room : A model driven approach. *In International Workshop on Ubiquitous Computing (IWUC 2006), Paphos, Cyprus*, volume 23, page 13–20, 2006. Cité pages 12 et 14.

NBDM : NBDM - modèle de données neutre pour la simulation de bâtiments, 2008. URL http://nbdm.org/. Cité pages 17 et 121.

Noriaki Kano, Nobuhiku Seraku, Fumio Takahashi et Shinichi Tsuji : Attractive quality and must-be quality. *Journal of the Japanese Society for Quality Control*, 1984. ISSN 0386-8230. Cité page 47.

OASIS : oBIX - open building information xchange. URL http://www.obix.org/. Cité page 17.

OMG : Reusable asset specification, novembre 2005. URL http://www.omg.org/spec/RAS/. Cité pages 54, 66 et 98.

OMG : OMG unified modeling language (OMG UML), superstructure, août 2011. Cité page 58.

Jason Pascoe, Nick Ryan et David Morse : Issues in developing context-aware computing. *In Handheld and ubiquitous computing*, page 208–221, 1999. URL http://link.springer.com/chapter/10.1007/3-540-48157-5_20. Cité page 12.

Robin R. Penner et Erik S. Steinmetz : Model-based automation of the design of user interfaces to digital control systems. *Systems, Man and Cybernetics, Part A : Systems and Humans, IEEE Transactions on*, 32(1):41–49, 2002. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=995528. Cité page 8.

Huy N Pham, Qusay H Mahmoud, Alexander Ferworn et Alireza Sadeghian : Applying model-driven development to pervasive system engineering. *In SEPCASE '07 : Proceedings of the 1st International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments*, page 7, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2970-4. Cité pages 12 et 14.

Pros~Labs : ProS labs - moskitt feature modeler. URL http://oomethod.dsic.upv.es/labs/index.php?option=com_content&task=view&id=51&Itemid=35. Cité page 65.

C. Pröglhöf, M. Schuss, K. Orehounig et A. Mahdavi : Implementation of a passive cooling strategy in an existing building using a simulation-based predictive control approach–a case study. 2011. Cité page 8.

R. Zach et A. Mahdavi : Monitoring for simulation validation. *BauSim 2010-Building Performance Simulation in a Changing Environment*, pages 190–195, 2010. Cité page 11.

R. Zach, M. Schuss, C. Pröglhöf, K. Orehounig, R. Bräuer et A. Mahdavi : An integrated architecture for energy systems and indoor climate monitoring in buildings. Osterreich, 2011. Cité page 11.

Roland Reichle, Michael Wagner, Mohammad Ullah Khan, Kurt Geihs, Jorge Lorenzo, Massimo Valla, Cristina Fra, Nearchos Paspallis et George A. Papadopoulos : A comprehensive context modeling framework for pervasive computing systems. *In Distributed applications and interoperable systems*, page 281–295, 2008. URL http://link.springer.com/chapter/10.1007/978-3-540-68642-2_23. Cité page 25.

Mark-Oliver Reiser et Matthias Weber : Multi-level feature trees. *Requirements Engineering*, 12(2):57–75, avril 2007. ISSN 0947-3602, 1432-010X. URL http://link.springer.com/article/10.1007/s00766-007-0046-0. Cité page 36.

M. Riebisch : Towards a more precise definition of feature models. *Modelling Variability for Object-Oriented Product Lines*, page 64–76, 2003. Cité pages 41, 45 et 46.

A. Rowe, M. E. Berges, G. Bhatia, E. Goldman, R. Rajkumar, J. H. Garrett, J. M. F. Moura et L. Soibelman : Sensor andrew : Large-scale campus-wide sensing and actuation. *IBM Journal of Research and Development*, 55(1.2):6 :1 –6 :14, mars 2011. ISSN 0018-8646. Cité page 10.

Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux et Yves Bontemps : Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, février 2007. ISSN 1389-1286. URL http://www.sciencedirect.com.gate6.inist.fr/science/article/B6VRG-4KXVB79-2/2/02f8747fe0169f2a97c5f9a7c0dd12c7. Cité pages 28, 33 et 42.

Sergio Segura, David Benavides, Antonio Ruiz-Cortés et Pablo Trinidad : Automated merging of feature models using graph transformations. *In Generative and Transformational Techniques in Software Engineering II*, page 489–505. Springer, 2008. URL http://link.springer.com/chapter/10.1007/978-3-540-88643-3_15. Cité page 36.

SINTEF : Common variability language wiki, 2012. URL http://www.omgwiki.org/variability/doku.php. Cité page 28.

Jim Steel, Keith Duddy et Robin Drogemuller : A transformation workbench for building information models. *In Proceedings of the 4th international conference on Theory and practice of model transformations*, ICMT'11, pages 93–107, Zurich, Switzerland, 2011. Springer-Verlag. ISBN 978-3-642-21731-9. Cité page 17.

Thomas Strang et Claudia Linnhoff-Popien : A context modeling survey. *In In : Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England*, 2004. Cité page 12.

R. Studer, V. R Benjamins et D. Fensel : Knowledge engineering : principles and methods. *Data & knowledge engineering*, 25(1-2):161–197, 1998. Cité page 18.

T. Bednasch, K. Czarnecki, U. W. Eisenecker et M. Lang : SourceForge.net : captain feature, 2003. URL http://captainfeature.sourceforge.net/. Cité pages 65 et 77.

Thein Tun et Patrick Heymans : Concerns and their separation in feature diagram languages : An informal survey. 2009. URL http://oro.open.ac.uk/33231/. Cité page 36.

Thibaut Possompès, Christophe Dony, Marianne Huchard et Chouki Tibermacine : Design of a UML profile for feature diagrams and its tooling implementation. *In Software Engineering and Knowledge Engineering (SEKE 2011)*, pages 693—698, Miami, FL, USA, 2011a. Cité pages 3 et 121.

Thibaut Possompès, Christophe Dony, Marianne Huchard et Chouki Tibermacine : Model-driven generation of context-specific feature models. *In Software Engineering and Knowledge Engineering (SEKE 2013)*, pages 250—255, Boston, MA, USA, 2013. Cité pages 3 et 121.

Thibaut Possompès, Christophe Dony, Marianne Huchard, Hervé Rey, Chouki Tibermacine et Xavier Vasques : Towards software product lines application in the context of a smart building project. *Proceedings of the 2nd International Workshop on Model-driven Product Line Engineering (MDPLE 2010)*, pages 73—84, 2010a. URL http://hal-lirmm.ccsd.cnrs.fr/lirmm-00494953/en/. Cité pages 3 et 121.

Thibaut Possompès, Christophe Dony, Marianne Huchard, Hervé Rey, Chouki Tibermacine et Xavier Vasques : A UML profile for feature diagrams : Initiating a model driven engineering approach for software product lines. *In Journée Lignes de Produits*, pages 59–70, France, 2010b. URL http://hal-lirmm.ccsd.cnrs.fr/lirmm-00542800/PDF/A_UML_profile_for_feature_diagrams.pdf. Cité pages 3 et 121.

Thibaut Possompès, François Briant, Christophe Dony, Marianne Huchard et Chouki Tibermacine : Diagramme de features, adaptation par transformation dans le contexte des bâtiments intelligents. *In Journée Lignes de Produits*, octobre 2011b. URL http://hal-lirmm.ccsd.cnrs.fr/lirmm-00635159. Cité pages 3 et 121.

Thomas von der Maßen et Horst Lichter : Deficiencies in feature models. *In Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, 2004. Cité page 33.

Thomas Thum, Don Batory et Christian Kastner : Reasoning about edits to feature models. *In Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 254–264, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. URL http://dx.doi.org.gate6.inist.fr/10.1109/ICSE.2009.5070526. Cité page 36.

J. C Trigaux et P. Heymans : Software product lines : State of the art. Rapport technique, Institut d'Informatique FUNDP, 2003. Cité page 2.

University~of~Waterloo : SPLOT - software product line online tools. URL http://www.splot-research.org/. Cité page 65.

Xavier Vasques, Thibaut Possompès, Hervé Rey, Marine Le Touzé, Nicolas Auboin, Emmanuelle Passot et Benoit Lange : Analysis and knowledge discovery from sensors data to improve energy efficiency. *In Proceedings of CIB-W78 W102 conference*, 2011. Cité pages 3, 6, 120 et 121.

W. Zhang, H. Zhao et H. Mei : A propositional logic-based method for verification of feature models. *Lecture Notes in Computer Science*, 3308:115–130, 2004. Cité pages 41, 42, 44 et 48.

Wei Zhang, Hong Mei et Haiyan Zhao : Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering*, 11(3):205–220, juin 2006. ISSN 0947-3602. URL http://www.springerlink.com/index/10.1007/s00766-006-0033-x. Cité page 41.

Abstract

Software product lines aim at reusing documents, source code, architectures, and, all artefact created during software development achieved in a given domain. Nowadays, we use “feature models” to facilitate the reuse of such elements. The approach consists in describing, in this feature model, artefacts and their usage constraints, and then to identify representative features for creating a new product. In some situations, a feature represents an artefact associated to a context element that must be handled by the product. Such a feature, and its related constraints, can be cloned for each occurrence of instances of this element in a given context.

In this thesis, we try to determine the impact of a product execution context on a future product features. We first explore different ways for representing feature models and a product context. Then, we propose a generic method to adapt a feature model to context elements.

This thesis has been achieved in the context of the RIDER project (Research for IT Driven EneRgy efficiency). This project aims at reducing energy waste due to an inappropriate management of energy sources and needs. The heterogeneousness of building equipments and each building specificities require to adapt energy optimisation software. We propose to apply a software product line approach to this project. More precisely, we propose to apply to this project our feature model context adaptation methodology, in order to adapt energy optimisation software to each building specific context.

Keywords: *Software product lines, Context adaptation, feature models, energy optimisation*

Résumé

Les lignes de produits logiciels ont pour objectif la réutilisation des documents, codes sources, architectures, et plus généralement tout artefact créé durant le développement de logiciels d'un même domaine. Pour cette réutilisation, on utilise aujourd'hui des "modèles de caractéristiques". L'approche consiste à décrire dans ce modèle les caractéristiques des artefacts créés et les contraintes permettant de les assembler, puis à sélectionner les caractéristiques représentatives d'un nouveau produit en le générant en tout ou partie. Dans certaines situations, une caractéristique représente un artefact associé à un élément du contexte que le produit doit gérer. Une telle caractéristique, et les contraintes relatives à sa mise en œuvre, peuvent être clonées pour chaque occurrence de l'élément dans le contexte.

Dans le cadre de cette thèse, nous cherchons à déterminer l'impact du contexte d'exécution d'un futur produit sur les caractéristiques d'une ligne de produits logiciels. Nous explorons tout d'abord les différentes manières de représenter un modèle de caractéristiques et le contexte d'un produit. Nous proposons ensuite une méthode générique pour adapter un modèle de caractéristiques aux éléments d'un contexte.

Cette thèse a été réalisée dans le contexte du projet RIDER (Reasearch for IT Driven EneRgy efficiency). Ce projet a pour objectif la réduction des pertes énergétiques subies à cause d'une gestion inappropriée des sources et des besoins énergétiques des bâtiments. La variété des équipements et les spécificités de chaque bâtiment nécessitent une adaptation au cas par cas des logiciels d'optimisation énergétique. Nous proposons donc d'appliquer à ce projet une approche par lignes de produits logiciels, et plus particulièrement, notre méthode d'adaptation de modèles de caractéristiques au contexte, pour adapter les logiciels d'optimisation énergétique au contexte spécifique de chaque bâtiment.

Mots clefs : *Lignes de produits logiciels, Adaptation au contexte, modèles de caractéristiques, Optimisation énergétique de bâtiments*

LIRMM — 161, rue Ada — 34095 Montpellier cedex 5 — France