INTÉGRATION DES ACTIVITÉS DE PREUVE DANS LE PROCESSUS DE DÉVELOPPEMENT DE LOGICIELS POUR LES SYSTÈMES EMBARQUÉS

Amine RAJI

These

TELECOM Bretagne

N° d'ordre : 2012telb0000

Sous le sceau de l'Université européenne de Bretagne

Télécom Bretagne

En habilitation conjointe avec l'Université de Bretagne Sud

École Doctorale – SICMA

Intégration des activités de preuve dans le processus de développement de logiciels pour les systèmes embarqués

---

Thèse de Doctorat

Mention : « Sciences et Technologies de l'Information et de la Communication »

Présentée par **Amine Raji**

Département : – Nom du département –

Laboratoire : UMR CNRS 3192 Lab-STICC

Directeur de thèse : Prof. Yvon Kermarrec Co-directeur : Mr. Philippe Dhaussy

Soutenue le XX mars 2012

Jury :

Rapporteurs :	M. Frédéric Boniol, Professeur, ONERA Toulouse
	M. Benoît Baudry, HDR, IRISA Rennes
Examinateurs :	Mme. Isabelle Borne, professeur, IUT Vannes, France
	M. Laurent Pautet, Professeur, Telecom PARIS
Directeurs :	M. Yvon Kermarrec, Professeur, Télécom Bretagne, France
	M. Philippe Dhaussy, Enseignant-Chercheur, Ensta-Bretagne Brest
Invités :	M. Antoine Beugnard, Professeur, Telecom Bretagne Brest

Abstract

In past years, formal verification techniques and tools were widely developed and used by the research community. However, the use of formal verification at industrial scale remains difficult, expensive and requires lot of time. This is due to the size and the complexity of manipulated models, but also, to the important gap between requirement models manipulated by different stackholders and formal models required by existing verification tools.

This dissertation aims therefore to develop a methodology that define activities that fill this gap by generating formal artifacts from textual requirements and existing design models. Our approach is based on previous work on the exploitation of contexts for formal verification, particularly, CDL language. We extended UML use cases with the ability to precisely describe interaction scenarios between the system under validation and its context. We also defined a requirement specification language based on the processing of natural language to formalize textual requirements. This formalization is performed thanks to model transformations that generate CDL properties from textual requirements and CDL context models form extended use cases scenarios. The proposed methodology is instantiated on a reel industrial case study provided by our industrial partner.

Résumé

En dépit de l'efficacité des méthodes formelles, en particulier les techniques d'analyse de modèles (*model checking*), à identifier les violations des exigences dans les modèles de conception, leur utilisation au sein des processus de développement industriel demeure limitée. Ceci est dû principalement à la complexité des modèles manipulés au cours de ces processus (explosion combinatoire) et à la difficulté de produire des représentations formelles afin d'exploiter les outils de vérification existants.

Fort de ce constat, les travaux exposés dans ce mémoire contribuent au développement d'un volet méthodologique définissant les activités conduisant à l'obtention des artefacts formels. Ceux-ci sont générés directement à partir des exigences et des modèles de conception manipulés par les ingénieurs dans leurs activités de modélisation. Nos propositions s'appuient sur les travaux d'exploitation des contextes pour réduire la complexité de la vérification formelle, en particulier le langage CDL. Pour cela, nous avons proposé une extension des cas d'utilisation UML, afin de permettre la description des scénarios d'interaction entre le système et son environnement directement dans le corps des cas d'utilisation. Aussi, nous avons proposé un langage de spécification des exigences basé sur le langage naturel contrôlé pour la formalisation des exigences. Cette formalisation est opérée par transformations de modèle générant des propriétés CDL formalisées directement des exigences textuelles des cahiers des charges ainsi que les contextes CDL à partir des cas d'utilisations étendus. L'approche proposée a été instanciée sur un cas d'étude industriel de taille et de complexité réelles développer par notre partenaire industriel.

Remerciements

Ce document représente le fruit de trois années de travail, je profite de cet instant pour remercier tous ceux qui ont contribué, de près ou de loin, à l'accomplissement de cette thèse.

Mes remerciements vont tout d'abord à Frédéric Boniol et Benoit Baudry pour avoir accepté de relire mon manuscrit et d'en être les rapporteurs.

Un éternel merci à Philippe Dhaussy. Merci pour ta confiance et pour avoir accepté d'encadrer mes travaux de thèse. Ton écoute, ton soutient sans limites et ta disponibilité ainsi que ton amitié ont fait de ces trois années de thèse une période inoubliable et si agréable. Par ailleurs, ta rigueur et tes conseils m'ont aidé à garder les objectifs de mes travaux en vue et les atteindre de la manière la plus optimale. Pour tout cela, je suis infiniment redevable.

Grand merci à Yvon Kermarrec qui à bien voulu diriger cette thèse et qui n'a ménagé aucun effort pour me faciliter ce travail de recherche. De par ses connaissances, ses conseils et sa disponibilité, il a très certainement contribué grandement à l'élaboration de ce travail.

Ce travail a aussi et avant tout été réalisé au sein d'une équipe de recherche dont les relations entre ses membres sont le moteur au quotidien de mes travaux. Ainsi, je veux remercier tous les collègues du département informatique de l'ENSTA-Bretagne pour leur accueil chaleureux. Je remercie Bruno Aizier, Pierre-Yves Pillain, Jean-Charles Roger et Lucas Le Roux pour les discussions intéressantes et fertiles en idées nouvelles. Merci à Dominique kerjean et Joël Champeau pour l'ambiance de travail bon-enfant qu'ils diffusent autour d'eux. Merci aussi à mes collègues de bureau Pierre Gauvillé et Jean-Philippe Schneider de m'avoir tenu compagnie tout le long de cette période.

Ces années de travail n'auraient pu être réalisées sans un soutien extérieur et infini de ma famille et de mes amis. Je remercie mes parents, mon frère Mounir et ma soeur Samira ainsi que toute ma famille de m'avoir soutenu pendant toutes ces années et pour leurs encouragements permanents. Grand merci à tous mes amis pour les repas, les soirées, les week-ends, les sorties, les matchs de foot et toutes les autres activités qui ont contribué à rendre ces années plus agréables.

Table des matière

Table des figures

Liste des tableaux

Publications

Les propositions de cette thèse ont fait l'objet de plusieurs publications nationales et internationales:

- Communications internationales (Avec édition d'actes et comité de sélection) :

  1. [Raji & Dhaussy 2011c] Amine RAJI, Philippe DHAUSSY, *Use Cases Modeling for Scalable Model-Checking*, In the 18th Asian Pacific Software Engineering Conference (APSEC), pages 65-72. University of Science, VNU-HCM, IEEE Computer Society, DOI10.1109/APSCEC.2.02101.12.525, Ho Chi Minh City, Vietnam, December 2011.

  2. [Raji & Dhaussy 2010b] Amine Raji, Phillipe Dhaussy, *User Context Models : a Framework to Ease Software Formal Verifications*, In 12th International Conference on Enterprise Information Systems, pages 380-383, Volume 3. ISAS, Funchal, Madeira Portugal, June 2010.

  3. [Raji & Dhaussy 2011b] Amine RAJI, Philippe DHAUSSY, *Improving formal verification practicability through user oriented models and context-awareness.* In Proceeding of the 8th International Workshop on Model-Driven Engineering, Verification and Validation (MODEVVA), Volume 4, pages 51–54, Wellington, New Zealand, October 2011.

  4. [Raji *et al.* 2010] Amine RAJI, Philippe DHAUSSY, Bruno AIZIER, *Automating Context Description for Software Formal Verification*, Inproceedings of the 7th Workshop on Model-Driven Engineering, Verification, and Validation (MODEVVA), pages 76-82, Oslo Norway, October 2010.

- Communications nationales (Avec édition d'actes et comité de sélection) :

  1. [Raji & Dhaussy 2011a] Amine RAJI, Philippe DHAUSSY, *Modèles orientés utilisateurs pour la vérification formelle en contexte industriel* 7èmes journées sur l'Ingénierie Dirigée par les Modèles (IDM), ISBM 978-2-917490-15-0, Lille, France, Juin 2011

  2. [Raji & Dhaussy 2010a] Amine RAJI, Philippe DHAUSSY, *Automatic Formal Model Derivation from Use Cases*, 6èmes journées sur l'Ingénierie Dirigée par les Modèles (IDM), pages 1–8, Pau, France Mars 2010.

INTRODUCTION ET BIBLIOGRAPHIE

TELECOM Bretagne

1 Introduction générale

1.1 Contexte de travail

1.1.1 Recherche de fiabilité des systèmes embarqués

Les systèmes informatiques embarqués connaissent depuis plusieurs années un essor considérable dans tous les domaines : automobile, aéronautique, espace, contrôle industriel, télécommunications. Les architectures logicielles doivent être conçues pour assurer des fonctions critiques soumises à des contraintes très fortes en termes de fiabilité et de performances temps réel. Mais malgré les progrès techniques, la grande taille de ces systèmes engendre l'introduction d'une plus grande gamme d'erreurs.

Les systèmes informatiques embarqués sont souvent soumis à des contraintes strictes imposées par leur environnement. Ils doivent répondre à des caractéristiques particulières qui rendent leurs analyses à la fois spécifique et pointue [Lee *et al.* 2007]. Parmi ces systèmes, les systèmes asynchrones, composés de sous-systèmes communiquant par échange de messages via des files d'attente (par exemple dans les systèmes et les protocoles de communication avionique bord/sol...), apportent une complexité encore supérieure. Celle-ci rend l'analyse des performances et la validation de ces systèmes difficile [Roychoudhury 2009]. En effet, leur analyse doit garantir que des aspects comme leur ordonnancement, le dimensionnement de leurs ressources, ou encore leurs comportements (détection d'interblocage, respect des contraintes temporelles) sont corrects.

Dans ce travail, nous nous limiterons à ce dernier aspect, c'est-à-dire la vérification du comportement des modèles logiciels. Dans ce mémoire, quand nous considérons des exigences, nous évoquons les exigences comportementales [Davis 1993], c'est-à-dire celles qui spécifient le comportement du système, celles-ci pouvant néanmoins référencer des contraintes temporelles ou non.

Actuellement, les industries engagent tous leurs efforts dans la mise en œuvre des processus de test et de simulation à des fins de certification. Néanmoins, les techniques utilisées deviennent rapidement contraignantes, voir inexploitables, pour découvrir des erreurs pouvant conduire à des situations catastrophiques. La couverture des jeux de tests s'amincie au fur et à mesure que la complexification des systèmes croit et il devient nécessaire d'utiliser de nouvelles méthodes pour garantir la fiabilité des logiciels. Il est

crucial de pouvoir recourir à des outils s'appuyant sur des méthodes fiables pour aider le développement de logiciels.

1.1.2 Les méthodes formelles

Les méthodes formelles ont contribué, depuis plusieurs années, à l'apport de solutions rigoureuses et puissantes pour aider les concepteurs à produire des systèmes non défaillants. Dans ce domaine, les techniques de *model-checking* [Queille & Sifakis 1982, Clarke *et al.* 1986] ont été fortement popularisées grâce à leur faculté d'exécuter automatiquement des preuves de propriétés sur des modèles logiciels. Elles permettent de confronter un modèle de conception aux propriétés à vérifier et ainsi détecter les erreurs de conception. Ces techniques de vérification peuvent être automatiques (presse-bouton) et connaissent un succès grandissant dans l'industrie depuis plusieurs années. Le principe consiste à construire, à partir du modèle du logiciel à valider, l'espace des états atteignables puis, en parcourant l'ensemble de ces états, à vérifier le respect des propriétés comportementales et ainsi que les contraintes temporelles attendues du système.

De nombreux outils (model-checkers) mettant en œuvre cette technique ont été développés [Holzmann 1997, Larsen *et al.* 1997, Berthomieu *et al.* 2004, Fernandez *et al.* 1996, Cimatti *et al.* 2000]. Malgré leur performance croissante, leur utilisation en contexte industriel reste encore difficile. Leur intégration dans un processus d'ingénierie industriel est alors encore faible comparativement à l'énorme besoin de fiabilité dans les systèmes critiques. Cette contradiction trouve en partie ses causes dans la difficulté réelle de mettre en œuvre des concepts théoriques dans un contexte industriel.

Une première difficulté liée à l'utilisation de ces techniques de vérification provient du problème bien identifié de l'explosion combinatoire du nombre de comportements des modèles, induite par la complexité interne du logiciel qui doit être vérifié. Cela est particulièrement vrai dans le cas des systèmes embarqués temps réel, qui interagissent avec des environnements impliquant un grand nombre d'entités. Beaucoup de travaux en méthodes formelles ont été menés afin de maîtriser l'explosion combinatoire [Mc.Millan & Probst 1992, Peled 1998, E.A. Emerson and S. Jha and D. Peled 1997, Alur *et al.* 1997, Valmari 1991, Godefroid *et al.* 1996, Bosnacki & Holzmann 2005, Park & Kwon 2006]. Des techniques développées autour du model-checking consistent en général à diminuer l'espace d'états du système global de telle manière à pouvoir effectuer une recherche d'accessibilité d'états.

Une autre difficulté est liée à l'expression formelle des propriétés nécessaires à leur vérification. Les artefacts produits à travers les activités du processus de développement industriel (exigences, spécification, modèles de conception) ne sont pas directement exploitables pour l'utilisation des outils de vérification. En effet, les exigences sont disponibles sous une forme textuelle qui les rendent inexploitables sans une réécriture préalable (formalisation) [Lu *et al.* 2008]. Traditionnellement, le model-checking implique de pouvoir exprimer les exigences à l'aide de formalismes de type logique temporelle comme LTL [Pnueli 1977] ou CTL [Clarke *et al.* 1986]. Bien que ces langages aient une

grande expressivité, il n'est pas aisé de les utiliser systématiquement pour la spécification de modèles de taille industrielle afin de pouvoir valider ces derniers.

L'approche, dans laquelle s'insère ce travail, prend un point de vue complémentaire, par rapport aux techniques décrites précédemment, en considérant la réduction non plus du système mais plutôt de son environnement.

1.1.3 L'approche OBP

Partant des constats précédents, des travaux [Roger 2006, Dhaussy & Boniol 2007, Dhaussy *et al.* 2009, Dhaussy *et al.* 2011] se sont penchés sur cette problématique en cherchant à faciliter l'utilisation des outils de model-checking. Les objectifs visés étaient d'étudier les conditions et les techniques permettant à un ingénieur en charge de vérifier des exigences sur un modèle logiciel dans un contexte industriel. La contribution voulait être un apport pour permettre d'exprimer facilement des exigences sous une forme compréhensible par un non expert des logiques temporelles et de pouvoir mener des vérifications par model-checking sur des modèles de grande taille.

L'idée suivie, pour contourner l'explosion combinatoire lors des explorations des modèles, est, d'une part, de chercher à réduire les comportements des modèles lors de leur exploration en considérant leur environnement. La réduction est basée sur une description de cas d'utilisation particuliers de l'environnement, nommés *contextes* avec lequel le système interagit. L'objectif est de guider le model-checker à concentrer ses efforts non plus sur l'exploration d'un automate global mais sur une restriction pertinente de ce dernier pour la vérification de propriétés spécifiques.

D'autre part, un langage à base de patrons de définition de propriété a été défini pour faciliter l'expression des exigences. La description des contextes et des propriétés est intégrée dans un même langage nommé CDL (*Context Description Language*) et exploité par l'outil OBP (*Observer-Based Prover*) [Dhaussy *et al.* 2009].

Nous décrivons plus en détail cette approche dans la section 3.2.

1.2 Limites de l'approche OBP/CDL

Suite aux expérimentations mener avec l'approche OBP/CDL, nous dressons un bilan et identifions les difficultés de mise en œuvre de celle-ci. Nous nous proposons ensuite d'étendre les travaux précédents pour contribuer à apporter des améliorations.

Lors des expérimentations, les diagrammes de contexte ont été construits par des ingénieurs eux-même en collaboration avec l'équipe qui a proposé CDL. Cette construction s'est effectuée d'une part, à partir de scénarios décrits dans les documents de conception et, d'autre part, à partir des documents d'exigences.

Deux difficultés majeures sont alors apparues. La première est le manque de description complète et cohérente du comportement de l'environnement des modèles analysés.

Par exemple, des données concernant les modes d'interaction peuvent être implicites. Le développement des diagrammes CDL requiert alors beaucoup de discussions avec les experts qui ont conçu les modèles afin d'expliciter toutes les hypothèses associées aux contextes. La deuxième difficulté concerne la compréhension des exigences et leur formalisation en propriétés formelles. Celles-ci sont regroupées dans des documents de niveaux d'abstraction différents, correspondants aux exigences système ou dérivées. L'ensemble des exigences analysées était rédigé sous forme textuelle. Certaines d'entre elles donnaient lieu à plusieurs interprétations différentes. D'autres faisaient appel à une connaissance implicite du contexte dans lequel elles doivent être prises en compte.

Il nous apparaît donc nécessaire de pouvoir offrir à l'utilisateur des moyens simple à mettre en œuvre permettant de spécifier de manière cohérente et complète les contextes ainsi que les exigences à vérifier. Le but recherché est que l'utilisateur puisse construire un ensemble d'artefacts nécessaires et suffisants pour produire automatiquement les modèles CDL.

Nous détaillons ici ces deux types de difficultés.

1.2.1 Difficulté liée à l'expression des contextes et des exigences

Dans le but d'éviter l'explosion combinatoire lors de la vérification avec l'approche OBP/CDL, il est nécessaire de pouvoir identifier une multitude de contextes suffisamment restrictifs pour limiter le nombre de comportements du modèle lors de l'exploration. Ceci implique une description d'un ensemble de contextes qui décrivent de manière complète les scénarios d'utilisation du système.

Toutefois, le langage CDL est considéré par les industriels comme étant de plus bas niveau. En effet, un modèle CDL référence des opérations de communication, en terme d'entrée et sortie de messages, dotés parfois de nombreux paramètres. Construire un modèle CDL revient donc à identifier l'ensemble des acteurs interagissant avec le système dans un contexte particulier, puis tous les événements échangés ainsi que leurs enchaînements. La construction de tels modèles n'est donc pas une tâche facile.

Les cas d'utilisations, fournis par les partenaires, sont généralement décrits en langage naturel, sous la forme de diagrammes d'activités ou de diagrammes de séquences. Ils décrivent les interactions entre les composants du système et les entités de l'environnement. Chaque diagramme, décrivant un cas d'utilisation référence plusieurs acteurs. Un modèle CDL formalise, quant à lui, le contexte en décrivant séparément le comportement de chaque acteurs. Aujourd'hui, la formalisation des interactions, décrits dans les cas utilisation, sous forme de modèles CDL est un processus manuel qui nécessite un effort important pour faire le lien entre ces deux niveaux de modélisation (Cas d'utilisation et CDL), en plus d'une bonne connaissance de la syntaxe et la sémantique de CDL. Il y a donc un fossé entre les descriptions fournies par les cahiers de charges et les documents de spécification et les modèles CDL (Figure 1.1). Ceci est dû à l'écart sémantique entre les descriptions textuelles des uses cases, décrivant les scénarios et les modèles CDLs qui capturent les échanges de messages émis ou reçus par chaque acteur.

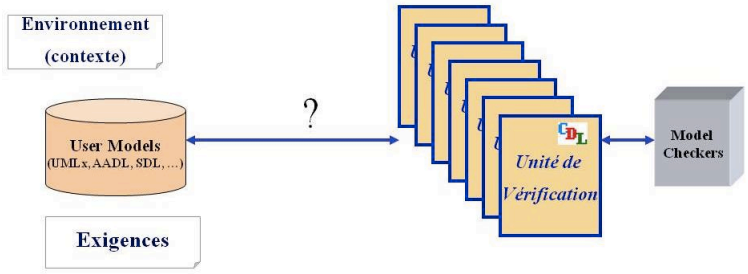

Figure 1.1 : Lien entre les modèles utilisateur et les modèles CDL

Par ailleurs, l'expression de exigences pose également certaines difficultés. En effet, les exigences exprimées dans les cahiers des charges sont décrites en langage naturel. Elles référencent des conditions liées au comportement du modèle, le comportement spécifique (un mode) du système, ainsi que les évènements dont certains n'étant pas toujours observables. Dans le langage CDL, les patrons de spécification de propriétés permettent de décrire les propriétés en terme de détection d'évènements (envois et réceptions de messages), de prédicats constituant les champs de vérification des propriétés (les Scopes) [Konrad & Cheng 2005a]). Là aussi, le passage entre les exigences textuelles et le ou les patrons à utiliser nécessite un effort important de réécriture.

1.2.2 Définition d'une méthodologie de vérification avec CDL

Toujours dans l'objectif d'aider l'utilisateur dans sa construction des modèles CDL. Il est nécessaire d'identifier clairement les étapes de cette activité. En effet, les modèles manipulés par celui-ci en phase de conception doivent pouvoir être exploités pour générer automatiquement les modèles CDL.

Dans ce contexte, nous ne prétendons pas apporter de solutions pour assurer la complétude de l'ensemble des cas d'utilisation que l'utilisateur identifie et formalise. Ceci ferait l'objet d'un travail spécifique. Mais nous cherchons à proposer un type de structure de données rassemblant l'ensemble des données nécessaires à la génération des modèles CDL.

1.3 Objectif de la thèse

L'objectif de cette thèse est de réduire l'écart entre les modèles manipulés par les ingénieurs au cours du processus de développement et les modèles CDL. Ceci dans le but de faciliter l'application des techniques de vérification par *model checking* sur les modèles de conception en contexte industriel. Pour atteindre cet objectif, nous allons adresser chacune des difficultés listées dans la section précédente.

1. *Génération des modèles comportementaux des acteurs de l'environnement:* Tout

d'abord, il est nécessaire de pouvoir faire la liaison entre, d'une part, les descriptions des uses cases d'un document de spécification et des modèles de conceptions et, d'autre part, les modèles comportementaux des acteurs. Une idée à explorer est de laisser l'utilisateur se concentrer sur la formalisation des cas d'utilisation et automatiser la construction des modèles CDL. Pour cela, nous proposons de spécifier ces contextes à un niveau d'abstraction plus élevé et de générer automatiquement les modèles CDL décrivant les comportements des acteurs de l'environnement. Nous proposons de faciliter la tâche à l'utilisateur en lui permettant de s'appuyer sur les cas d'utilisation qu'il a l'habitude de rédiger.

2. *Formalisation des exigences des cahiers des charges :* Nous proposons de spécifier les exigences à un niveau d'abstraction plus élevé et de générer automatiquement les propriétés CDL.

3. *Éléments méthodologiques:* Toutefois, cet effort de formalisation de contextes et des exigences, pour la vérification du modèle du système étudié, doit s'intégrer aux différentes phases du processus de développement. Cette intégration ne peut s'opérer qu'à travers la définition d'une méthodologie facilitant la manipulation des artefacts nécessaires à l'utilisation des outils de vérification formelle le long du processus de développement.

   Nous identifions une structure, que nous nommons UCM (User Context Model) (Figure 1.2), pour regrouper toutes les données exploitables par des transformations pour générer automatiquement les modèles CDL. En suite, nous proposons un cadre méthodologique définissant les activités permettant la construction des structures UCM le long du processus de développement. Nous avons donc proposé des éléments d'ordre méthodologique pour faciliter la mise en œuvre des modèles UCM.

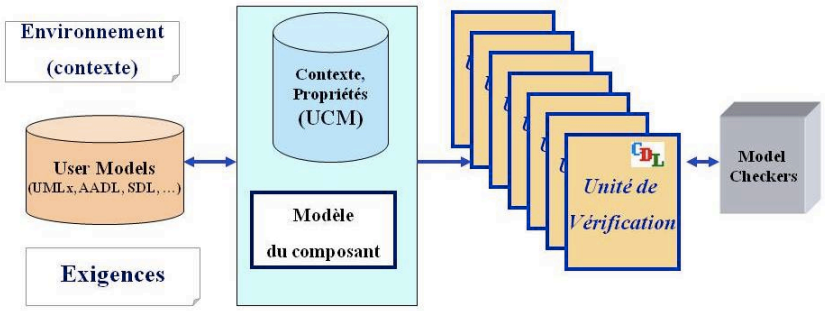

Figure 1.2 : Composant UCM

En effet, l'exploitation des structures UCM au sein d'un développement industriel a besoin d'être encadrée par une méthodologie adaptée aux processus existants et aux artefacts utilisés lors des différentes activités. Celle-ci doit permettre de capitaliser les modèles de contextes formalisés dans les UCM afin de pouvoir les réutiliser pour de nouvelles vérifications lorsque le modèle ou les exigences évoluent. Ainsi, cette capitalisation des modèles

formels ne pourrait avoir lieu sans une méthodologie précisant le processus de construction des artefacts formels à travers le processus de développement utilisé.

Les travaux développés dans cette thèse portent sur le formalisme UCM. Celui-ci est proche des formalismes que l'utilisateur a l'habitude de manipuler, tels que les diagrammes UML et les exigences textuelles. Les modèles UCM permettent la génération automatiquement des modèles CDL. Pour cela, la définition des UCM s'articule autour de trois formalismes, proposés sous la forme de DSLs[1]: Un premier DSL nommé *XUC* (eXtend User Context) pour la description formelle des contextes. Un deuxième DSL nommé *URM* (User Requirement Model) pour l'expression des exigences. Enfin, un DSL nommé *DSpec* pour la description du domaine de données.

Le formalisme XUC :

Ce langage est dédié à la capture et à la formalisation des contextes. Ce dernier permet aux ingénieurs de décrire les contextes de leurs systèmes sous une forme qui soit, à la fois, proche des modèles qu'ils ont l'habitude de produire et compilable par une machine. Il permet la capture des interactions Système/Environnement, dans les différents cas d'utilisation.

Le formalisme URM :

Ce formalisme, basé sur le langage naturel structuré et contraint, a pour but l'expression des exigences textuelles sous une forme exploitable par une machine. La motivation d'un tel formalisme est de pouvoir transformer automatiquement ces exigences en propriétés basées sur les patrons de définition de CDL. Les modèles de contextes XUC référencent les exigences exprimées dans le formalisme URM. Un mécanisme de lien est prévu entre les propriétés formalisées URM et les modèles de contextes XUC. Ce lien permet de regrouper des ensembles de propriétés à traiter lors du traitement des scénarios.

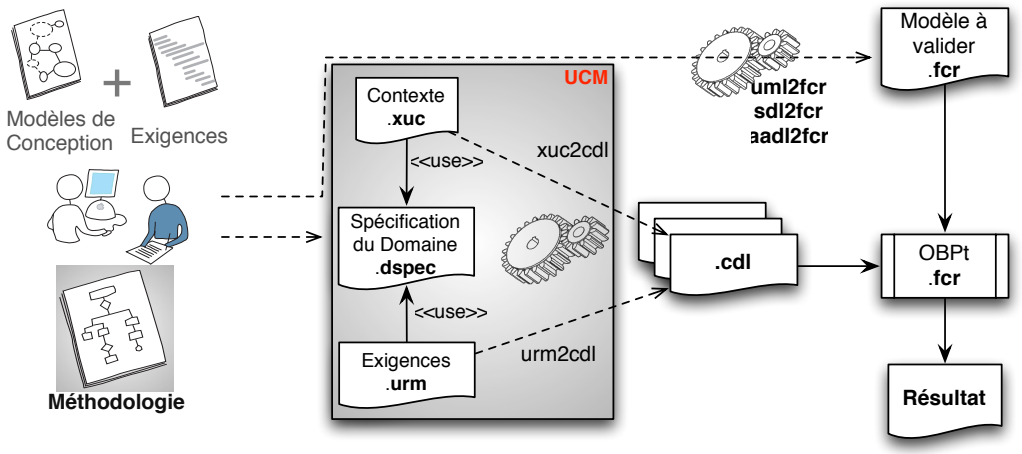

Figure 1.3 : Composant UCM pour la génération de modèles CDL

[1]Domain Specific Language

Le formalisme DSpec :

Pour réaliser ce *"mapping"* entre UCM et CDL, les langages XUC et URM s'appuient sur une base de connaissance du domaine du système étudié. Cette base de connaissances, regroupée dans un dictionnaire de donnée, vise à préciser la sémantique des notions et entités utilisées dans la description des contextes et des exigences.

Ainsi, ces trois formalismes sont mis en œuvre dans une chaîne de transformations vers les modèles CDL pour lesquelles nous avons spécifié les règles (figure 1.3). De cette façon, un UCM constitue l'artefact central pour la constitution d'un guide méthodologique d'utilisation des langages XUC, URM et DSpec à travers les différentes phases aboutissant à produire les modèles CDL.

1.4 Organisation du mémoire

Cette thèse est organisée en quatre parties principales comme le montre la figure 1.4.

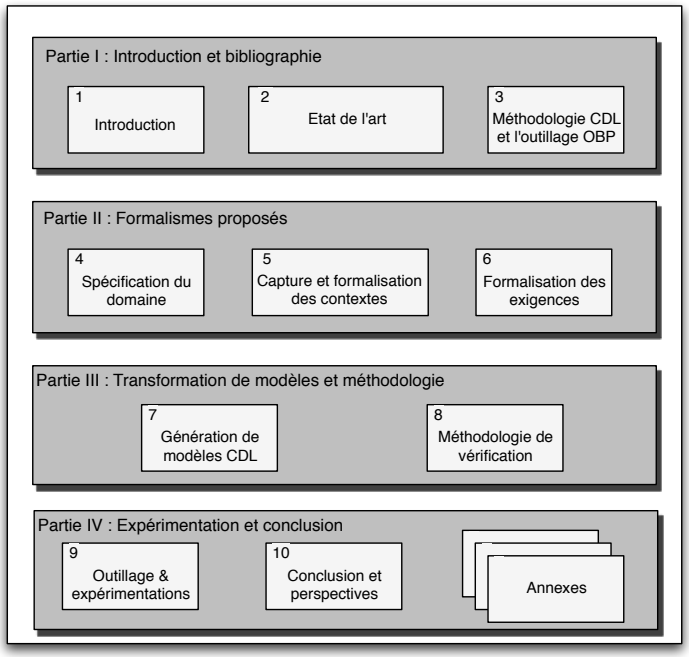

Figure 1.4 : Organisation globale du mémoire

La première partie fournit un état de l'art des travaux en relation avec cette thèse et se termine par une présentation du langage CDL et de la chaîne d'outils OBP. Le chapitre 2 propose une vue générale des travaux autour de la formalisation des contextes et des exigences, des problématiques liées à cette activité et insiste sur les formalismes utilisés et les techniques de vérification employés. Le chapitre 3 décrit le cas d'étude utilisé le long de ce mémoire pour illustrer les concepts proposés puis présente le langage CDL et l'outillage OBP. Il fournit un exemple de spécification de modèles CDL (exigences et contextes) et montre, exemple à l'appui, la difficulté de produire et mesurer la cohérence

d'une telle spécification opérationnelle pour un modèle de taille industrielle.

La deuxième et la troisième partie présentent nos contributions. La deuxième partie détaille les trois formalismes proposés avec les descriptions de leurs syntaxes et sémantiques respectives. Le chapitre 4 commence par présenter notre langage de spécification des entités liées au domaine. Spécification sur laquelle s'appuie les langages CDL et URM présentés, respectivement, aux chapitres 5 et 6. La troisième partie est consacrée à la présentation de notre méthodologie de vérification. Celle-ci est basée sur les formalismes présentés dans la deuxième partie ainsi que les algorithmes de génération de modèles CDL présentés dans le chapitre 7.

Dans la quatrième et dernière partie, on décrit les expérimentations menées sur le cas d'étude industriel dans le chapitre 9. Puis, le chapitre 10 présente la conclusion générale de ce mémoire et dresse un bilan de nos contributions. Il trace aussi un panorama des principales pistes de recherche et perspectives qui se dégage à partir de nos travaux. À la fin de cette partie, nous regroupons l'ensemble des annexes.

2 Etat de l'art

Nous présentons dans ce chapitre l'état de l'art relatif aux concepts qui nous permettent de mettre en œuvre nos propositions. Ces concepts sont relatifs à la formalisation des scénarios d'interaction entre le système et son contexte, à la formalisation des exigences ainsi qu'aux méthodologies de développement. Nous présentons les différentes approches de descriptions de méthodologies à partir desquelles nous nous inspirons pour identifier celle que nous proposons dans ce mémoire. Celle-ci a pour finalité de conduire l'activité de vérification des exigences au sein du processus de développement industriel.

2.1 Capture et formalisation des entités du domaine

La spécification des entités et concepts d'un domaine d'application à pour objectif de pouvoir les référencer dans les modèles d'exigences et de conceptions. Cette idée a été introduite par Kaindl dans [Kaindl 1997]. Traditionnellement, les méthodes d'analyse et de conceptions orientées objet sont utilisées pour identifier ces entités dans un domaine particulier afin de les utiliser à des fins de conceptions. Une discussion sur ces incompréhensions est présentée dans [Kaindl 1999]. En effet, l'auteur explique les difficultés liées au passage des modèles d'analyse aux modèles de conceptions et présente un certain nombre de recommandations pour limiter leurs impacts. Parmi ces recommandations, il précise l'intérêt de définir une spécification claire des entités et des concepts d'un domaine particulier.

Une séparation claire entre les artefacts liés à l'analyse des exigences et ceux nécessaires à la conception et à l'implémentation est présentée sous la forme d'un métamodèle [Ebner & Kaindl 2002]. La figure 2.1 présente le métamodèle en question.

Dans le reste de cette section, nous présentons une approche typique d'analyse et de conception orientée-objet pour découvrir comment les entités du domaine sont identifiées et utilisées. Par la suite, nous exploitons ces informations pour définir un langage adapté à la capture des entités de la spécification du domaine dans le chapitre 4.

2.1.1 Méthodologie d'analyse et de conception orientée-objet

Nous résumons ici une méthodologie type pour l'analyse et de conception orientée-objet [Booch 2004] (*OOAD*[1]). Celle-ci est schématisée par la figure 2.2. Dans la suite de cette section, nous allons détailler les activités de chacune de ces trois étapes. Cette

[1] *Object-Oriented Analysis and Design*

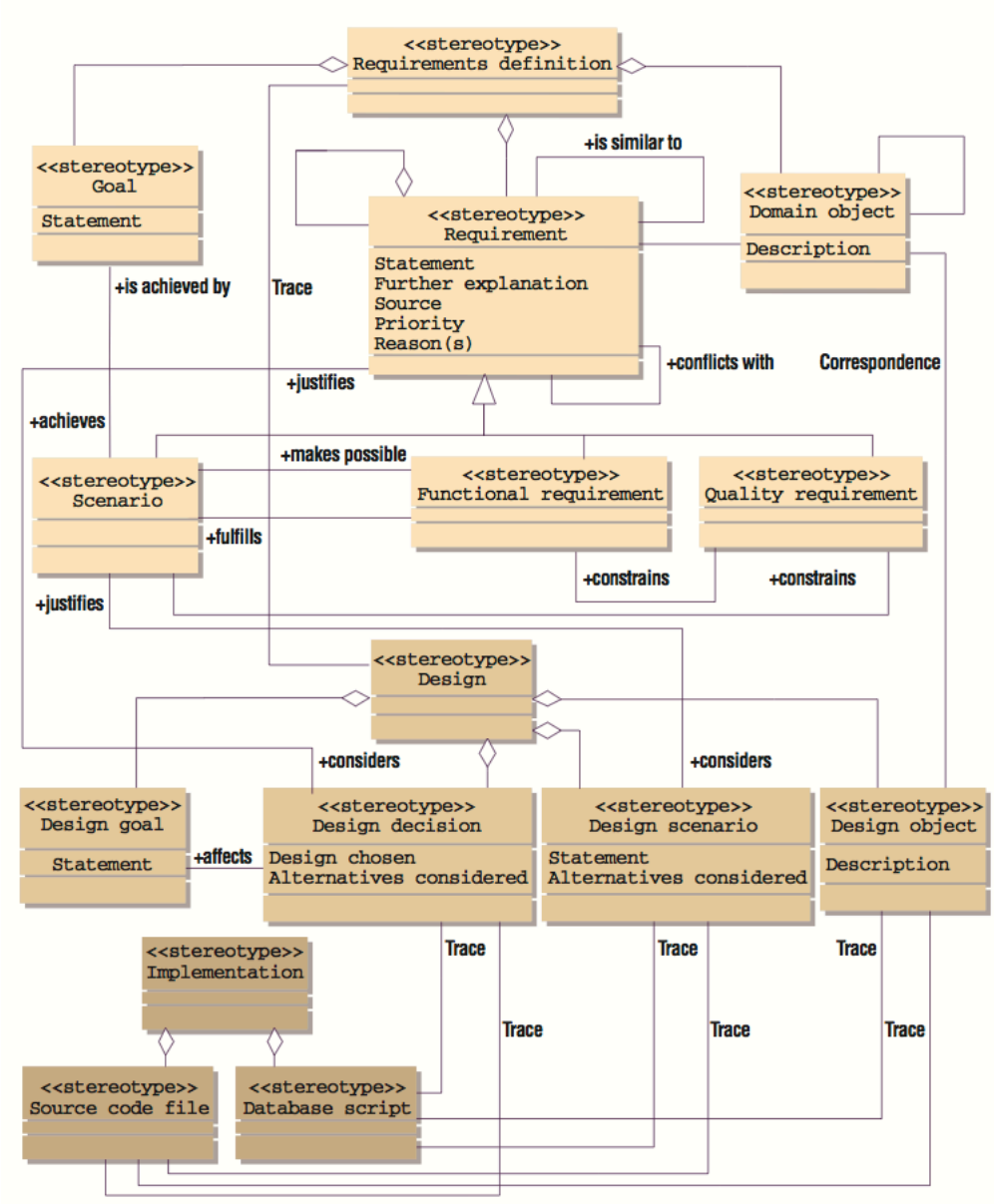

Figure 2.1 : Métamodèle pour représenter les artefacts liés aux exigences, à la conception et à l'implémentation et les relations entre eux [Ebner & Kaindl 2002]

description va nous permettre de dégager les éléments méthodologiques qui vont nous servir par la suite pour décrire notre méthodologie de vérification par exploitation des contextes présentée au chapitre 8.

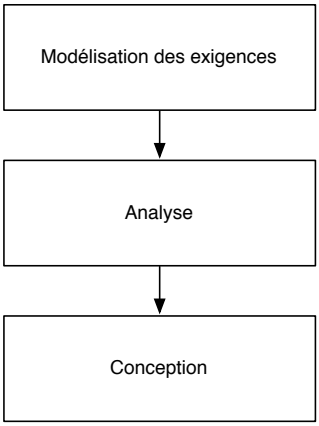

Figure 2.2 : Trois grandes étapes d'une méthodologie de conception et d'analyse OO

2.1.1.1 Modélisation des exigences

Entrée: les cas d'utilisations, les cahiers des charges, autres artefacts liés au domaine métier

Objectif: Décomposer les artefacts du domaine métier pour capturer et définir le cadre et les responsabilités du système à construire afin de répondre aux exigences.

Activités:

- Identifier les principaux objectifs métier, les processus et les ressources
- Écrire les descriptions des cas d'utilisations avec une identification claire des *acteurs* et les *données* échangés entre le *système* et son *environnement* et exprimer les prés/posts conditions et des invariants.
- Produire un diagramme de use case avec l'ensemble des cas d'utilisations et leurs relations.

2.1.1.2 Analyse orientée-objet

Entrée: Tous les artefacts produites par la phase de la modélisation des exigences

Objectif: Analyser les artefacts exigences afin de construire le modèle du domaine

Activités:

- Relier les cas d'utilisation entre elles. Ceci permet de produire en un premier niveau de décomposition du domaine sous la forme de domaines fonctionnalités qui serons déclinés en sous-systèmes. Montrer la décomposition du diagramme des cas d'utilisations en utilisant la notation des packages en UML.

- Pour chaque sous-système, extraire les classes, les attributs et leurs relations en se basant sur les descriptions du cas d'utilisations. Montrer cette décomposition (aussi appelé *modèle du domaine*) en utilisant une notation semblable à celle des diagrammes de classe UML.

- Extraire les concepts communs aux différents sous-systèmes afin de les référencer dans leurs modèles du domaine respectif.

- Préciser les relations entre les entités du modèle du domaine, ces entités permettrons, avec les cas d'utilisation, de construire les diagrammes d'interactions.

- En se basant sur les cas d'utilisation, les modèles de domaine de chaque sous-système et les diagrammes d'interaction du système globale, définir pour chaque sous-système un diagramme d'interaction afin d'identifier leurs interfaces.

- Avec les cas d'utilisation, les modèles du domaine et les diagrammes d'interactions des sous-systèmes, développer un diagramme d'interaction des objets. L'objectif de cette étape est de capturer:

  * comment les objets du domaine collaborent pour accomplir la fonctionnalité décrite par le cas d'utilisation
  * les définitions des interfaces des objets
  * les associations et les interactions ainsi que les séquences d'interactions entre les objets

- En utilisant tous les diagrammes d'interactions, construire une Diagramme de Collaboration Unifié (*DCU*) sans les numéros des messages, les multiples d'objets du même type ou les noms des objets.

- En utilisant le *DCU*, enregistrer chaque message en tant qu'une méthode dans le modèle du domaine.

- Construire une machine à état pour chacun des objets. Les messages du *DCU* sont les principales sources des événements pour les transitions.

Notes: Il est aussi recommandé de capturer les invariants, les préconditions et les postconditions pour chaque entité identifiée au cours de l'analyse orientée-objet. Chaque entité doit être prise en compte avec les objectifs métier et reliée à ses contraintes et aux exigences non fonctionnelles ainsi qu'à tous les artefacts capturés durant la phase de la modélisation des exigences.

2.1.1.3 Conception orientée-objet

Entée: tous les exigences et les artefacts du modèle du domaine (modèle du domaine, *DCU*, machines à états).

Objectif: Créer un modèle de conception orienté-objet à partir du modèle du domaine en prenant en compte les exigences du système et les ressources de développement.

Activités:

- Construire une architecture de haut niveau du système en se basant sur les informations issues des modèles du domaine des différents sous-systèmes et les exigences liées à l'architecture. Définir les interfaces du système et des sous-systèmes.
- En utilisant le modèle du domaine et le *DCU* ainsi que les exigences liées à l'architecture du système, *mapper* les concepts du domaine aux classes de l'application logiciel. Ce mapping doit prendre en compte les contraintes de conceptions, telles que la réutilisabilité, et la maintenance, ainsi que les exigences internes au système, telles que la persistance, la sécurité, les performances. . . .
- Définir l'architecture opérationnelle du système, les composants, les processus et l'allocation des ressources de calcul, les canaux de communications, les interfaces et les protocoles.
- Pour chaque entité opérationnelle (composant, processus, canal de communication):
  * définir de quels objets est elle construite,
  * faire une distinction entre les objets actifs, tels que les contrôleurs et les collaborateurs [Kaindl 1997] et objets passifs comme les données.
- Raffiner les interfaces des classes
- Pour chaque classe, définir ses algorithmes internes, classes additionnelles, les types de données, les attributs, etc.

Les étapes décrites ci-haut représentent un exemple d'une méthodologie typique de conception et d'analyse orientée-objet. Il existe différentes méthodologies dans la littérature, mais la conceptualisation du domaine métier au cours des premières phases est une activité commune à ces méthodologies. Les concepts orientent la spécification et ont un impact significatif sur tous les artefacts produits tout au long de l'analyse et de la conception.

Dans une perspective de vérification formelle des modèles de conceptions produits, la définition d'une spécification précise du domaine métier s'est avérée d'une importance capitale pour faire le lien entre les exigences et les contextes pertinents de leurs vérifications. Nous présentons dans le chapitre 4 notre langage pour capturer les entités du domaine et leurs relations afin de garantir la cohérence entre les exigences et les contextes de leurs vérifications. Ces contextes représentent les configurations du système à vérifier

de garantir une maîtrise sur la taille de l'espace d'état généré par l'outil de vérification formel utilisé.

La section qui suit présente un état de l'art sur ces contextes et leurs intérêts pour la vérification formelle des systèmes embarqués.

2.2 Les contextes

2.2.1 Intérêt de la formalisation des contextes pour la vérification formelle

Le besoin de disposer des moyens permettant la formalisation précise des contextes est d'autant plus ressenti dans un contexte industriel. En effet, de par la complexité intrinsèque des systèmes industriels et de par les communications le plus souvent asynchrones, les ingénieurs ne possèdent pas une cartographie complète de la dynamique de l'environnement de leur système; c'est-à-dire qu'ils ne peuvent pas déterminer en amont l'ordre d'émission ou de réception de chaque message entre l'environnement et le système. Par conséquent, le système est modélisé de manière à accepter plusieurs entrelacements possibles des acteurs de son environnement.

Un système réactif interagit avec son environnement à travers des actions (envois/réception de messages/événements, variables partagées...) afin de répondre à ses sollicitations. Ces actions, appelées *actions libres*, permettent à un système d'effectuer des échanges avec son environnement. Un système qui fonctionne sans interagir avec son environnement, c'est-à-dire sans actions libres, est un système *clos*. Par opposition, si le système interagit avec son environnement, il est appelé système *ouvert*. Une analyse d'accessibilité effectuée sur un système ouvert ne permet pas d'obtenir l'ensemble des comportements de ce système. De ce fait, l'auteur dans [Roger 2006] explique comment exploiter l'information sur le comportement de l'environnement d'un système et analyser son comportement dans des configurations spécifiques. Ces configurations spécifiques sont appelées *contextes* et font référence aux comportements attendus des acteurs de l'environnement vis-à-vis du système étudié.

Ainsi, la réduction par exploitation du contexte s'opère comme suit, l'automate du système à vérifier est composé avec celui représentant les comportements des différents acteurs de l'environnement par le biais d'un produit de synchronisation. Soit un système S d'alphabet Σ défini par $S = A_1 \parallel_{L_1} \cdots \parallel_{L_{n-1}} A_n$ avec A_i des automates temporisés et L_i des produits de synchronisation. Une action libre sur S est une action présente sur S mais n'est liée à aucun produit de synchronisation, c'est à dire, $a \in \Sigma \setminus \bigcup_{\forall i} L_i$. Soit Σ_L l'ensemble des actions libres. La définition d'un contexte est donnée comme suit:

> **Définition 1 (Contexte) :** Soit S un système ouvert. Un contexte C fermant S est un automate temporisé défini par $(Q, q_{init}, \Sigma_L, \rightarrow)$, tel que le produit $S \parallel_{\Sigma_L} E$ est un système clos.

Soit un système S et un contexte C. L'évolution de S fermé par C est donnée par la relation:

$$< (C, B_1)|(s, B_2) > \quad \rightarrow^a_\sigma S \quad < (C', B_1')|(s', B_2') >$$

Pour exprimer que S dans l'état s associé à un buffer d'entrée B_2 est fermé par le contexte (C, B_1) évolue vers l'état s' en recevant l'événement a (produit par le contexte) et en produisant la séquence d'événements σ. La fermeture du système par son contexte est définie par les règles suivantes:

- Si S peut produire σ, alors S évolue est σ est placé en fin du buffer de C.

$$\frac{(s, B_2) \quad \rightarrow_\sigma S \quad (s', B_2')}{< (C, B_1)|(s, B_2) > \quad \longrightarrow_\sigma^{null_e} S \quad < (C, B_1.\sigma)|(s', B_2') >} \quad \text{[Comp1]}$$

- Si C peut émettre a, C évolue et a est placé en fin du buffer de S.

$$\frac{(C, B_1) \quad \rightarrow^{a!} S \quad (C', B_1')}{< (C, B_1)|(s, B_2) > \quad \longrightarrow_{null_\sigma}^{a} S \quad < (C', B_1')|(s, B_2.a) >} \quad \text{[Comp2]}$$

- Si C peut consommer a, alors il évolue tandis que S reste inchangé.

$$\frac{(C, B_1) \quad \rightarrow^{a?} \quad (C', B_1')}{< (C, B_1)|(s, B_2) > \quad \longrightarrow_{null_\sigma}^{null_e} S \quad < (C', B_1')|(s, B_2) >} \quad \text{[Comp3]}$$

La composition de fermeture entre un système et son contexte est assimilable à une composition parallèle asynchrone: les comportements de C et de S sont entrelacés et la communication est réalisée par échanges asynchrones à travers des buffers.

Pour résumé, la réduction par exploitation de contextes consiste à réduire l'espace d'état du système en précisant les comportements de l'environnement avec lequel le système interagit. L'objectif est de guider le *model-checker* à concentrer ses efforts non plus sur l'exploration de l'automate global du système, mais sur une restriction pertinente de ce dernier pour la vérification de propriétés spécifiques.

Dans les sections suivantes, nous présentons les formalismes proposés dans la littérature pour permettre aux utilisateurs la capture et la formalisation des contextes. Puis, nous pointons les difficultés qui limitent l'exploitation de ces formalismes dans le cadre d'une vérification formelle. Enfin, nous proposons un langage dédié, facilitant aux utilisateurs la capture et la formalisation des contextes dans le cadre de la vérification formelle de modèles dans un contexte industriel.

2.2.2 Etat de l'art sur la capture et la formalisation des contextes

2.2.2.1 Les Cas d'utilisation: "Use Cases"

Depuis leur introduction, les cas d'utilisation sont devenus une méthode de choix pour la capture des exigences des systèmes logiciels [Whittle 2007]. Un cas d'utilisation tel qu'il a été défini par Cockburn [Cockburn 2000] est une description du comportement du système placé sous diverses conditions alors qu'il répond aux sollicitations de l'une des parties prenantes.

Un cas d'utilisation est représenté sous forme d'une combinaison d'un diagramme UML de cas d'utilisation [Booch *et al.* 1999], et d'un texte structuré sous la forme d'un canevas. Différents canevas ont été proposés, ces derniers présentent la séquence principale des

étapes qui définissent le cas d'utilisation ainsi que les séquences supplémentaires permettant de capturer les exceptions, les alternatives ainsi que les extensions. Les cas d'utilisation sont présentés, dans la majorité des cas, sous une forme informelle. Il n'y a pas de consensus sur la sémantique des diagrammes des cas d'utilisations et celle du texte des canevas proposés n'est pas spécifiée par le standard UML.

Le caractère non formalisé des cas d'utilisation facilite leurs utilisations, mais constitue une limitation à l'application des techniques d'analyse automatisées tels que la génération des cas de tests, la simulation, la validation, etc. Plusieurs travaux ont proposé d'introduire une sémantique rigoureuse aux descriptions des cas d'utilisation. Ces travaux peuvent être classés en deux catégories: la première s'est intéressée à la définition des restrictions structurelles sur le texte utilisable dans les canevas [Williams *et al.* 2005, Michal 2005], tandis que la deuxième catégorie propose le développement d'une sémantique formelle pour les éléments constituants les diagrammes de cas d'utilisation [Övergaard & Palmkvist 2004, Stevens 2001]. Les travaux existants sur la formalisation du texte contenu dans un cas d'utilisation définissent une grammaire pour un sous ensemble des mots du langage naturel qui peut être complétée par des définitions provenant d'un dictionnaire. Les approches qui proposent de définir une sémantique formelle pour les cas d'utilisation visent à spécifier formellement les différentes constructions proposées par UML telles que la sémantique des relations *«include»* et *«extend»* [Stevens 2007].

2.2.2.2 "Use Case Charts"

Whittle présente dans [Whittle 2007] un formalisme à trois niveaux qu'il a appelé *Use Cases Charts* (UCC). Les UCCs sont basées sur une extension des diagrammes d'activités UML afin de spécifier les cas d'utilisation en détail à des fins de simulation. Dans les UCCs, un cas d'utilisation est considéré comme étant un ensemble de scénarios où chaque scénario représente le comportement attendu du système ou une trace de son exécution. Ainsi, les fonctionnalités d'un système peuvent être données sous la forme d'un ensemble de cas d'utilisation où chaque cas d'utilisation est décrit par un ensemble de scénarios. La figure 2.3 présente le principe de composition des UCCs en trois niveaux.

En effet, les UCC permettent de spécifier les scénarios d'un cas d'utilisation sous la forme d'une description à trois niveaux: Le niveau-1, appelé *use case chart* est une extension du diagramme d'activité UML dans lequel chaque noeud est un use case; le niveau-2, appelé *scenario chart*, est un diagramme d'activité où chaque noeud est un scénario; le niveau-3 est un ensemble de diagrammes d'interactions UML2.0 [OMG 2007]. Ainsi, chaque cas d'utilisation du niveau-1 est défini par un ensemble de scénarios du niveau-2 connectés entre eux, et chaque scénario du niveau-2 est défini par un diagramme d'interaction UML2.0. Dans la figure 2.3, sept cas d'utilisation connectés dans le niveau-1 qui commence par un cas d'utilisation initial puis passe aux 4 suivant en parallèle. Chacun de ces sept cas d'utilisation est défini par un *scenario chart* du niveau-2. Seul le *scenario chart* du noeud pointé par une flèche en pointillés est montré par la figure 2.3. Celui-ci contient trois noeuds dont chacun sera défini par un diagramme d'interaction UML2.0 au niveau-3.

La sémantique des UCCs est définie sous la forme de traces d'exécutions. Le flux de contrôle d'un UCC commence par le noeud initial du niveau-1. Le flux passe ensuite aux différents noeuds du même niveau suivant les transitions du diagramme d'activité du

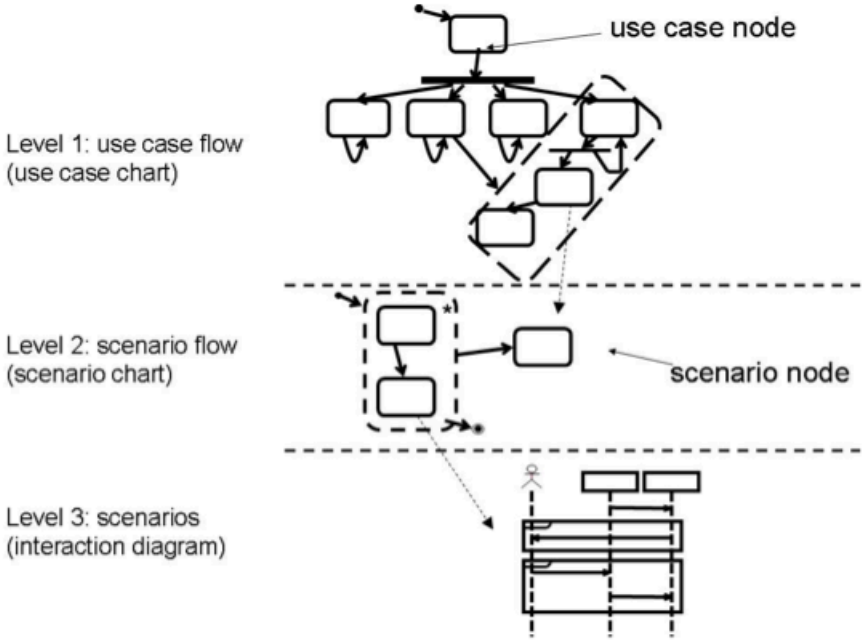

Figure 2.3 : Use Case Charts [Whittle 2007]

niveau-1. Lorsque le flux atteint un noeud d'un cas d'utilisation, le scénario du niveau-2 définissant ce cas d'utilisation est exécuté (le flux commence aussi par son noeud initial et termine lorsqu'il atteint le noeud final). La sémantique de chaque scénario est similaire à celle des hMSCs (*high-level message sequence charts* [ITU-TS 1999]).

2.2.2.3 "Use Case Maps"

Buhr et son équipe ont développé une notation appelée *Use Case Maps* permettant la description de scénarios à un niveau d'abstraction plus élevé sous la forme d'une séquence de *responsabilités* liés à un ensemble de composants [Buhr 1998]. Les *Use Case Maps* ont été proposés pour permettre aux concepteurs de se concentrer sur les aspects les plus pertinents du fonctionnement critique d'un système. Ils permettent de faire le lien entre les descriptions de haut niveau du comportement des systèmes et leurs architectures à l'aide d'une syntaxe graphique adaptée.

Les scénarios dans les *use case maps* sont représentés à un niveau d'abstraction au-dessus du niveau d'échange des messages entre les composants sous une forme centrée sur le chemin de fonctionnement afin d'augmenter le niveau de réutilisabilité des scénarios.

Les *Use Cases Maps* sont en cours de standardisation permettant la représentation explicite des relations entre les use cases sous une forme permettant d'abstraire les détails des scénarios. Un *use case map* représente les cas d'utilisation sous la forme d'un diagramme avec des chemins montrant la progression du scénario le long du cas d'utilisation. Les *use case maps* sont d'un niveau d'abstraction supérieur à celui des UCC et ne montrent pas les détails de chaque cas d'utilisation.

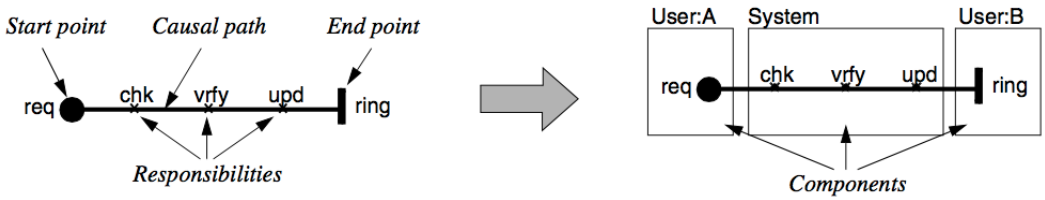

Figure 2.4 : Exemple d'un *use case maps* [Hassine 2005]

La figure 2.4 représente une connexion téléphonique simplifiée modélisée avec des *use cases maps* (*UCMaps*). Sur cette figure, le *UCMaps* n'est pas relié à une architecture spécifique, il est appelé *unbound UCMaps*. Plusieurs architectures alternatives peuvent être développés pour le même *Maps*. Cette séparation permet aux concepteurs de ne pas se limiter à des choix architecturaux précoces au cours du cycle de développement.

La figure 2.5-(a) montre le chemin défini par la figure 2.4 lié à deux utilisateurs connectés à travers une architecture à base d'agents. La figure 2.5-(b) présente un *Message Sequence Chart* (MSC) équivalent au *UCMaps* de la figure 2.5-(a).

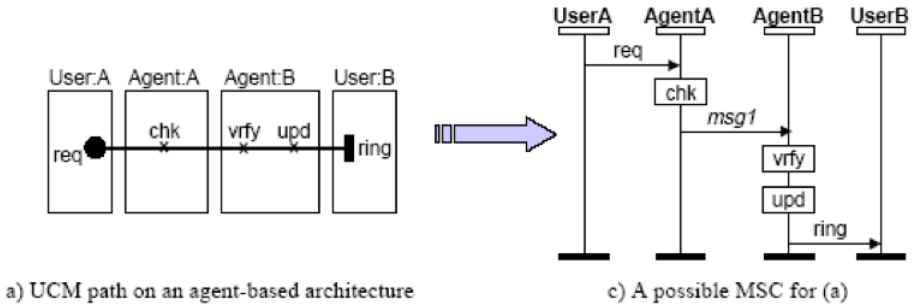

Figure 2.5 : (a) Chemin d'un *UCMaps* sur une architecture à base d'agents (b) un MSC possible pour (a) [Hassine 2005]

2.2.2.4 Discussion

Bien que plusieurs travaux existants se sont penchés sur la problématique de la formalisation des cas d'utilisation afin d'automatiser leur analyse, leur application dans un contexte industriel demeure limitée [Cockburn 2000]. Ceci est dû principalement au manque de méthodologies complètes allant jusqu'à la génération de code formel utilisable par les outils de vérification formelle existants. Au cours de nos travaux, nous nous sommes inspirés des deux catégories des travaux cités précédemment (section 2.2.2.1) pour définir une extension des cas d'utilisation UML. Cette extension vise à formaliser les canevas utilisés pour décrire les scénarios d'un cas d'utilisation et la définition de la sémantique formelle des concepts et relations qui composent un diagramme de cas d'utilisation UML.

En effet, dans un contexte industriel, il est commun de trouver des descriptions, sous forme de scénarios, accompagnant les cas d'utilisation. La relation précise en-

tre ces deux concepts est donnée de façon informelle [Uchitel *et al.* 2004]. La littérature propose différentes vues sur les relations entre les scénarios et les cas d'utilisation. Dans [Texel & Williams 1997] les cas d'utilisations sont considérés comme "des représentations des fonctionnalités attendues du système" alors que "un scénario est une description des étapes requises pour compléter un cas d'utilisation". Selon Jacobson [Jacobson 2004], les scénarios sont des "instances" des cas d'utilisations; ils décrivent les interactions entre les objets alors que les cas d'utilisations décrivent les interactions entre les différents modèles d'objets. Ainsi, Uchitel [Uchitel 2003] considère que les cas d'utilisations sont plus abstraits que les scénarios. De plus, il est admis dans la littérature que plusieurs scénarios peuvent être définis pour le même cas d'utilisation pour montrer les différentes façons de le réaliser. Les cas d'utilisations montrent une vue externe sur le système, séparant les acteurs du système en développement. Ainsi, les cas d'utilisation ne doivent pas, selon Jacobson [Jacobson 2004], révéler l'architecture interne du système en développement. Cette tâche est laissée aux diagrammes de séquences puisqu'ils donnent une description détaillée des différents composants interagissant au sein du système pour produire une fonctionnalité annoncée dans un cas d'utilisation.

2.3 L'INGÉNIERIE DES EXIGENCES

L'ingénierie des exigences (IE) concerne les méthodes et techniques permettant de produire une spécification d'exigences qui soit à la fois exhaustive, cohérente et reflétant effectivement les besoins à l'origine d'un projet logiciel. L'IE considère comme principal indicateur du succès d'un projet logiciel le niveau d'adéquation entre les exigences identifiées et les besoins réels ayant motivé le projet [Nuseibeh & Easterbrook 2000]. L'IE vise à améliorer le processus de production des exigences et l'élaboration des documents de spécification qui constituent le point d'entrée pour de nombreux ingénieurs participant à l'élaboration d'une solution. A titre d'exemple, on peut citer les travaux autour de la définition du concept d'objectif [van Lamsweerde & Letier 2004, van Lamsweerde 2008] qui permet une formalisation précise des exigences, fonctionnelles et non fonctionnelles. Dans le même esprit, l'OMG à normalisé SysML [OMG 2008b] introduisant les diagrammes d'exigences pour formuler les exigences afin d'assurer leur suivi dans le cycle de développement du système.

Un processus d'ingénierie des exigences constitue la partie amont du processus de développement logiciel. Ce dernier guide et accompagne le passage des besoins informels exprimés par les différentes parties prenantes d'un projet en une spécification d'exigences abstraite, appelée *spécification des besoins*. Cette spécification passe par un raffinement progressif jusqu'à l'obtention d'une *spécification d'exigences* qui va constituer le *cahier des charges*. Ce dernier est le point d'entrée pour les phases de développement en aval.

La production d'une telle spécification est une activité importante et difficile dans un projet de développement logiciel. En effet, la spécification des exigences est souvent présentée sous forme d'une collection de spécifications partielles et hétérogènes, résultat d'un travail d'équipe impliquant des parties prenantes, dont les rôles, les intérêts et les compétences sont très hétérogènes au sein du projet logiciel [van Lamsweerde 2000, Cheng 2007]. Elles sont partielles, car chacune des parties prenantes a un point de vue singulier de

ce que doit être le futur logiciel. Le caractère hétérogène est dû au fait que ces spécifications sont décrites à l'aide de langages variés adaptés à la variabilité des préoccupations.

Cependant, assurer que les exigences de haut niveau sont correctement reflétées dans les exigences de bas niveau et en suite dans les éléments de conceptions puis dans les composants d'implémentation est un aspect essentiel pour la production d'applications logicielles sûres [Nikora & Balcom 2009]. De nombreuses méthodes outillées ont été proposées pour capturer et analyser formellement les exigences. Toutefois, l'exploitation de ces méthodes à l'échelle industrielle demeure limitée [Craigen *et al.* 1995, Rosenblum 1996, Finkelstein & Kramer 2000, Konrad & H C Cheng 2005, Cheng 2007]. En effet, les cahiers de charges ainsi que les modèles de conception doivent être représentés à l'aide de langages de spécification formels afin de pouvoir utiliser ces techniques d'analyse. Malgré que ces techniques se soient avérés efficaces pour identifier les erreurs de conception, leur taux d'utilisation dans les processus de développement industriels est faible. Un effort important est nécessaire pour:

- apprendre un langage de spécification formel,
- exploiter un outil de vérification formel,
- acquérir le savoir-faire nécessaire à l'abstraction des détails qui ne sont pas nécessaires lors de la spécification des exigences dans un langage formel.

2.3.1 Capture, évolution et raffinement des exigences

Cette section présente un état de l'art des techniques utilisés pour la manipulation, l'évolution et la compréhension d'une spécification d'exigences. L'IE propose la décomposition et le classement de l'information afin de décomposer le problème initial en sous problèmes plus simples pour mieux gérer la complexité [Jackson 2000].

Le développement d'un système logiciel fait appel à plusieurs personnes, ou groupes de personnes —chacun avec sa propre perspective sur le système conduit par ces centres d'intérêt, ces connaissances, ses responsabilités ainsi que ces engagements — interagissant directement ou indirectement avec ce système [Sommerville & Sawyer 1997]. D'où l'émergence du besoin de considérer les différentes perspectives, communément appelées *Vues*, pour l'identification des exigences du système.

L'utilisation de la notion des *vues* comme approche au sein de l'ingénierie des Exigences (*Requierement Engenieering RE*) est apparue avec les travaux de Mullery[Mullery 1979]. Ce dernier présente une méthode d'expression des exigences collectées pour l'ingénierie des exigences en partitionnant le système en plusieurs *vues*, chacune liée à une partie prenante du système. En effet, une approche basée sur la notion de *vues* part du principe que toutes les informations nécessaires pour la spécification d'un système ne peuvent être découvertes à partir d'une seule perspective (vue). Depuis, les vues ont été utilisées pour identifier, organiser, modéliser et valider les exigences d'un système à logiciel prépondérant à partir de plusieurs perspectives.

2.3.2 Langages de modélisation des exigences

En contexte industriel, les exigences sont souvent exprimées dans des documents d'exigences sous la forme de texte écrit en langage naturel. Cela permet de faciliter la communication entre les différentes parties prenantes pour la validation des informations produites. Néanmoins, le langage naturel demeure informel et ambigu ce qui rend difficile l'analyse formelle de ses expressions. En effet, seule une expression opérationnelle des exigences disposants d'une sémantique précise est analysable à travers un processus outillé d'IE. Or, une exigence exprimée en langage naturel peut être sujette à beaucoup d'interprétations de la part des parties prenantes impliquées dans le projet, ce qui est source d'incohérences difficilement détectables.

Pour réduire les ambiguïtés inhérentes au langage naturel, beaucoup de travaux ont proposés d'exploiter ces caractéristiques dans un processus de formalisation des exigences en le couplant à une approche précise pour définir ce qu'on appelle un langage naturel contraint [Videira & Rodrigues da Silva 2005]. Nous présentons une discussion sur le langage naturel contraint à la section 2.3.3.

Plusieurs langages dédiés à la spécification opérationnelle des exigences ont été proposés dans la littérature ayant chacun leurs particularités. Les avantages de ces langages sont bien identifiés surtout pour la vérification formelle des systèmes critiques [Ryan 1992, Fabbrini *et al.* 2002, Fabbrini *et al.* 2002, Denger *et al.* 2003]. Dans [Hassine *et al.* 2010], l'auteur présente une classification et une comparaison détaillée de treize langages à base de scénarios selon onze critères d'évaluation.

La constatation qu'on peut dégager de cette étude comparative, c'est que différents langages et notations sont proposés dans la littérature afin de permettre une meilleure expressivité des exigences. Toutefois, la plupart des notations utilisées en IE pour décrire formellement les exigences ne sont pas toujours accessibles aux parties prenantes ce qui limite leurs implications [Heitmeyer 1998].

Dans ce contexte, des patrons de spécification d'exigences ont été proposés pour offrir une syntaxe plus accessible. Les langages à base de patrons de spécification sont moins expressifs que les langages présentés plus haut, mais sont plus accessibles à l'ensemble des parties prenantes. Ils participent à l'implication des parties prenantes dans la rédaction et la validation d'une spécification opérationnelle. Ils favorisent aussi le transfert technologique vers l'industrie des outils de vérification formelle [Brottier 2010]. La section suivante détaille l'utilisation des patrons de spécification des exigences ainsi que les travaux qui se sont intéressés à l'utilisation du langage naturel contraint afin de faciliter la transition entre la spécification des besoins et la formalisation des exigences opérationnelles.

2.3.3 Langage naturel contraint

L'utilisation du langage naturel pour spécifier des exigences a pour but de pouvoir profiter de l'aspect simple et universel qui le caractérise en le couplant à une approche précise (pas forcément formelle) afin de définir ce qu'on appelle *un langage naturel contraint*

[Videira & Rodrigues da Silva 2005]. Dans un langage naturel contraint, les mots utilisés représentent un sous-ensemble du langage naturel courant, où ces derniers gardent leur sens habituel, mais peuvent aussi être interprétés dans un contexte spécialisé (à travers des outils). Toutefois, utilisation d'un ensemble limité de construction du langage naturel réduit l'expressivité du langage contraint. Ce manque d'expressivité est remplacé par le gain en accessibilité proposé par ce type de langages. En effet, l'implication des parties prenantes dans le processus de définition et formalisation des exigences est importante dans la mesure où la description d'une spécification opérationnelle est plus accessible [Broy & Slotosch 2001].

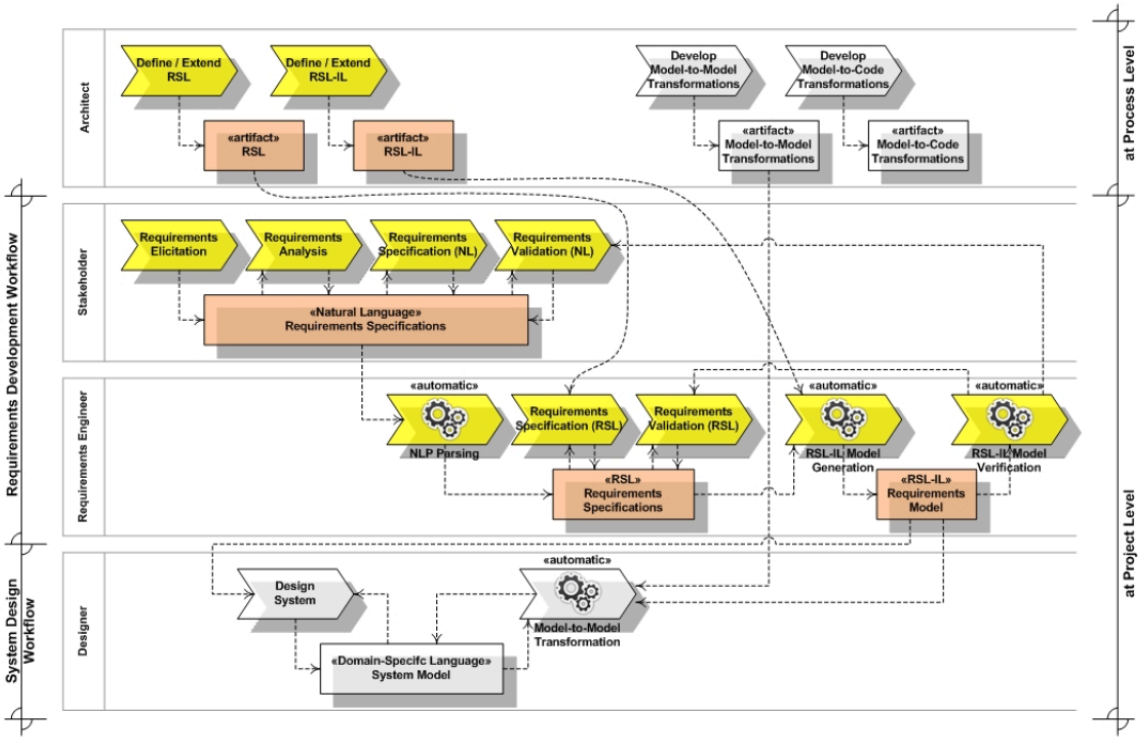

Figure 2.6 : Processus de formalisation des exigences selon Ferreira *et al.* [Ferreira & Rodrigues da Silva 2009]

Dans ce contexte, plusieurs travaux ont exploré la piste de spécification d'exigences à l'aide de langages naturels contraints et de patrons de spécification. Videira *et al.* [Videira *et al.* 2005] présentent le projet Project IT-RSL, proposant une approche pour l'utilisation du langage naturel dans la spécification des exigences des systèmes logiciels. Ce projet constitue une première étape d'un processus visant la génération de modèles UML et de code en se basant sur une approche de développement orientée modèle. L'approche proposée utilise des patrons pour la spécification des exigences des systèmes interactifs dont la combinaison permet la spécification de propriétés plus complexes. Après l'identification des patrons de propriétés, les auteurs présentent un métamodèle pour permettre la représentation des patrons identifiés. Une grammaire est proposée pour définir des règles de *mapping* des concepts de ce métamodèle sur des expressions

linguistiques. Ces expressions sont soumises à des tests de validation. L'approche telle qu'elle est définie dans [Videira & Rodrigues da Silva 2005] ne supporte que les exigences fonctionnelles. Toutefois, le métamodèle réalisé inclut des métas-classes pour des exigences non fonctionnelles et autres exigences pour un développement futur. La figure 2.6 illustre le processus de formalisation des exigences proposées par Videira *et. al* [Ferreira & Rodrigues da Silva 2009].

2.3.4 Patrons de propriétés

La grammaire utilisée pour décrire les propriétés en langage naturel est basée sur les patrons de spécification de propriétés proposées par Dwyer *et al.*[Dwyer *et al.* 1999] (figure 2.7). Ces patrons ont été formalisés dans cinq formalismes pour exploiter les outils de vérification formelle existants (CTL, LTL, GIL, INCA et QRE). Les patrons de Dwyer justifient leur efficacité pour la formalisation des propriétés par une étude effectuée sur 500 spécifications de propriétés formelles et qui a démontrée que les 92% des spécifications rencontrées pouvaient être capturées par un des patrons. Pour définir un patron de propriété, Dwyer propose en plus de la description et de l'ensemble de formules logiques qui la formalise, un champ appelé "champ d'application de la propriété". Ce champ décrit l'espace temporel durant lequel la propriété capturée doit être vraie. La figure 2.8 présente ces champs d'application.

OCCURRENCE	Absence	A given state/event does not occur within a scope
	Existence	A given state/event must occur within a scope
	Bounded Existence	A given state/event must occur k times, at least k times or at most k times within a scope
	Universality	A given state/event occurs throughout a scope
ORDER	Precedence	A given state/event must always be preceded by a given state/event within a scope
	Response	A given state/event must always be followed by a given state/event within a scope
	Chain Precedence	A given sequence of states/events must always be preceded by a given sequence of states/events (generalization of Precedence Pattern).
	Chain Response	A given sequence of states/events must always be followed by a given sequence of states/events (generalization of Response Pattern).

Figure 2.7 : Classification des patrons de spécification de propriétés de Dwyer [Dwyer *et al.* 1999]

Ainsi, le scope "Globally" exprime que la propriété doit être vérifiée le long de tout le séquencement d'états. Le scope "Before R" exprime que la propriété doit être vérifiée le long d'un séquencement d'état avant que l'état "R" ne soit atteint. Dans le scope

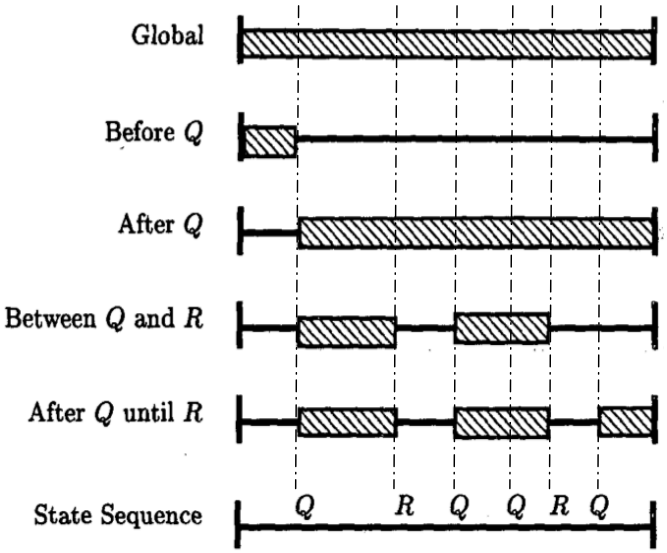

Figure 2.8 : Champs d'application de propriétés [Dwyer *et al.* 1999]

"After Q", la propriété doit être vérifiée dans l'état "Q" et après. "Between Q and R", la propriété doit être vérifiée le long de l'intervalle formé par l'état "Q" à gauche et l'état dont le successeur est l'état "R". Le scope "After Q until R" exprime la même chose que le scope "Between Q and R" à l'exception que l'intervalle peut ne pas être fermé à droite (car aucun état "R" n'est atteint dans le futur).

Dans [Konrad & Cheng 2005a], Cheng et Konrad proposent une extension des patrons de Dwyer en ajoutant des constructions pour exprimer des propriétés temporelles temps réels. À l'instar des patrons de Dwyer, ceux proposés par Cheng visent à formaliser les propriétés les plus rencontrées pour les systèmes temps réel. Ces patrons utilisent d'autres formalismes adaptés à l'expression des propriétés temps réel (MTL, TCTL, RTGIL).

La figure 2.9 montre l'extension des patrons de spécification de propriétés de Dwyer pour le domaine temps-réel (les patrons à l'extérieur du rectangle gris).

Concernant l'intégration des patrons de spécification des propriétés aux processus de développement, Konrad et Chang proposent dans [Konrad & H C Cheng 2006] un processus d'ingénierie supportant la spécification de modèles UML en utilisant un outil de modélisation UML pour analyser des propriétés spécifiées à l'aide du langage naturel et par la suite, faire le raffinement du modèle afin d'éliminer les erreurs découvertes lors de l'analyse.

Ce processus a été implémenté dans un outil, appelé SPIDER[2], permettant aux utilisateurs de spécifier et analyser un modèle UML en respectant les propriétés comportementales spécifiées en langage naturel. Pour faciliter le processus de spécification, les auteurs proposent une approche supportant l'instanciation automatique de propriétés en langage naturel à partir de *templates* avec des informations extraites du

[2]Specification Pattern Instantiation and Derivation EnviRonment

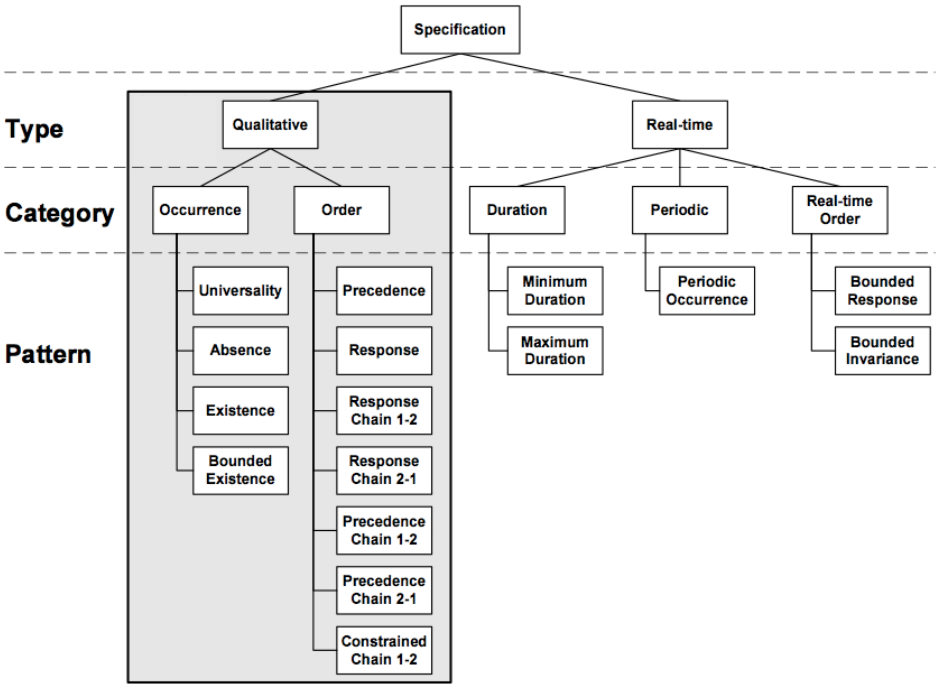

Figure 2.9 : Classification des patrons de propriétés

modèle considéré. Concrètement, ce processus est configuré pour analyser des modèles UML 1.4 sous la forme de fichiers XMI 1.1 afin générer des spécifications formelles en langage PROMELA pour le modèle *checker* SPIN[Holzmann 2004]. La figure 2.10 illustre le processus de spécification et d'analyse des exigences selon [Konrad & H C Cheng 2006].

2.4 LES MÉTHODOLOGIES DE DÉVELOPPEMENT

2.4.1 Processus de développement

Le cycle de développement d'un système embarqué décrit la manière dont sont structurées les différentes étapes menant à la réalisation du produit final à partir des exigences exprimées par le client [Feiler & Humphrey 1993]. Il existe plusieurs cycles de développement (cycle séquentiel, itératif, en V, en Y ...). Dans la mesure où nous cherchons à améliorer l'intégration des activités de vérification aux processus de développement existants, nous nous intéressons plus particulièrement aux mécanismes proposés pour produire les artefacts nécessaires à la conduite des activités de vérifications: Gestion des exigences, modélisation du système et la vérification formelle. Nous allons présenter deux exemples de processus de développement largement utilisés pour les projets de développement de logiciels; le cycle de développement selon la méthode UP [Booch *et al.* 1999] et le cycle de développement en V.

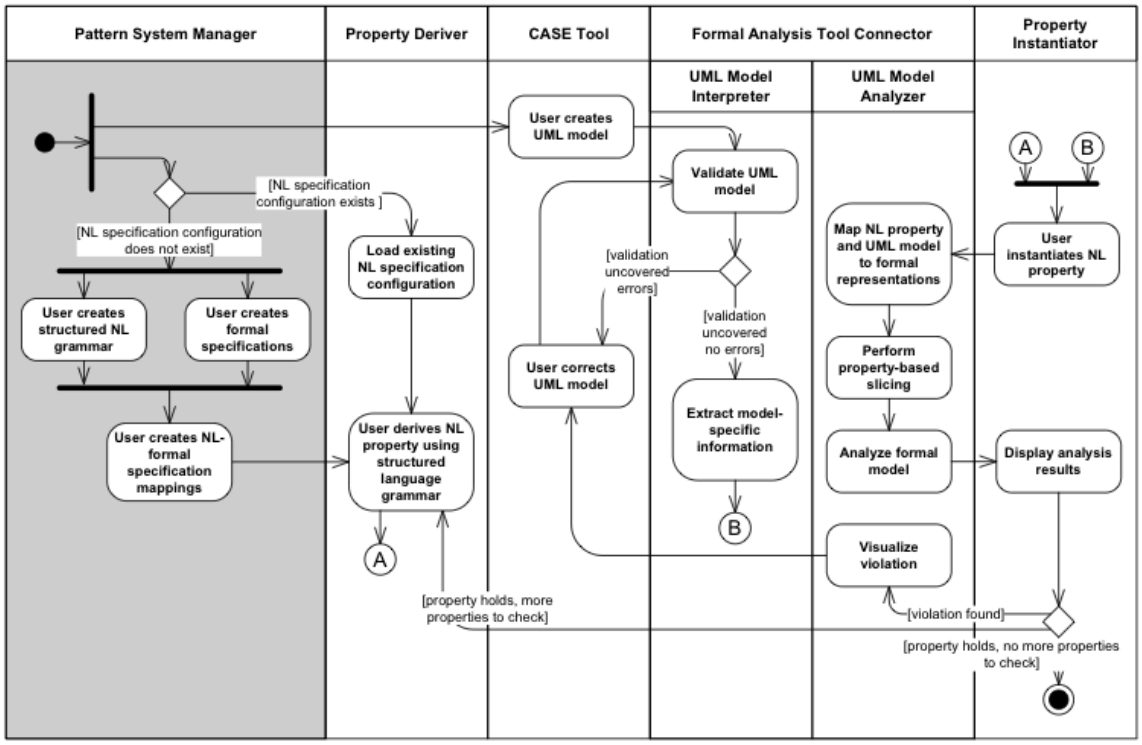

Figure 2.10 : Diagrammes d'activité UML illustrant le processus de spécification et d'analyse des exigences selon [Konrad & H C Cheng 2006]

2.4.1.1 Le modèle de développement selon la méthode UP

La méthode de type UP s'impose souvent dans des grands projets (*Rational Unified Process*) et est construite autour du langage UML [OMG 2007]. Selon [Booch *et al.* 1999], un processus unifié (UP) est "itératif et incrémental, centré sur l'architecture, construite à partir des cas d'utilisations et pilotée par les risques". La figure 2.11 montre le cycle itératif, découpé en 5 phases, de la méthode UP.

- *Le Recueil des besoins* permet de définir les besoins et faire la distinction entre les besoins *fonctionnels*, permettant la construction des modèles, et les besoins *non fonctionnels* qui forment la liste des exigences à satisfaire.
- *l'Analyse* a pour objectif de comprendre les besoins et les exigences afin de produire les spécifications des modèles d'analyse. Les modèles UML correspondants à cette spécification sont les cas d'utilisation pour une spécification complète des exigences fonctionnels ainsi que leurs structures sous forme de scénarii (diagrammes de séquences et d'interactions).
- *La Conception* est l'activité où sont approfondis les connaissances des différents composants du système ainsi que leurs principales interfaces (diagrammes de classes). Ceci prépare la phase d'implémentation en décomposant le système en plusieurs sous-composants qui correspondent aux différentes classes du système.

- L'*implémentation et le déploiement* concerne la planification et l'intégration des composants pour chaque itération et la production des classes et des sous-systèmes sous forme de code source. Le déploiement est décrit par les diagrammes de déploiement UML.
- Les *test* permet de confronter l'implémentation aux cas de tests planifiés à chaque itération à partir de la phase du recueil des besoins.

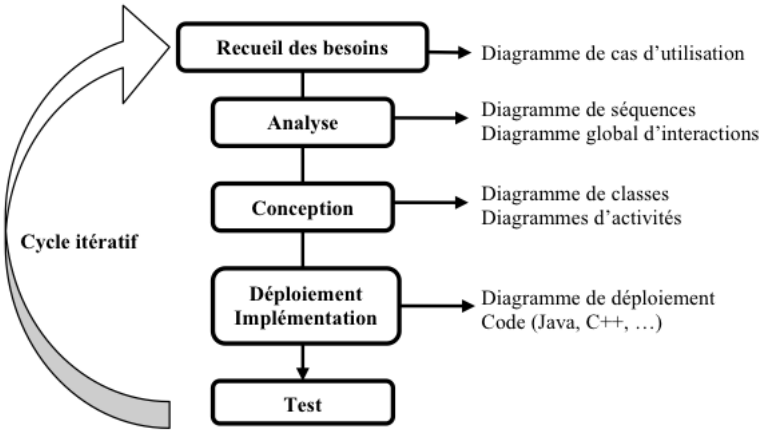

Figure 2.11 : Cycle de développement selon la méthode UP

2.4.1.2 Le modèle de développement en V

Le modèle de développement en "V" est l'un des modèles de développement les plus utilisés dans le domaine de développement de logiciels [Broy & Slotosch 1999]. Il s'agit d'un standard ISO permettant de structurer le processus de développement de logiciels. Son nom est inspiré du schéma global du procédé qui est composé de plusieurs étapes détaillées. La partie descendante du V représente la déclinaison du système en développement depuis les exigences jusqu'au code d'implémentation tandis que la partie ascendante représente les activités du contrôle de qualité (test des composants, leurs intégrations et le test du système global). La figure 2.12 illustre la place de la vérification au sein du modèle de développement en V proposé par [Broy & Slotosch 1999].

2.4.2 Processus de vérification formelle des exigences en contexte industriel

Dans [Fontan 2008], l'auteur présente plusieurs méthodes de conceptions utilisant le langage UML pour le développement de logiciels pour les systèmes embarqués temps réels distribués. Le survol de ces méthodes de conception basées sur le langage semi-formel UML, largement utilisé dans le milieu industriel, nous amène à faire le constat que la validation du système est basée principalement sur le test du code. Ceci est regrettable, car des fautes de conception dues à un défaut de validation peuvent être découvertes

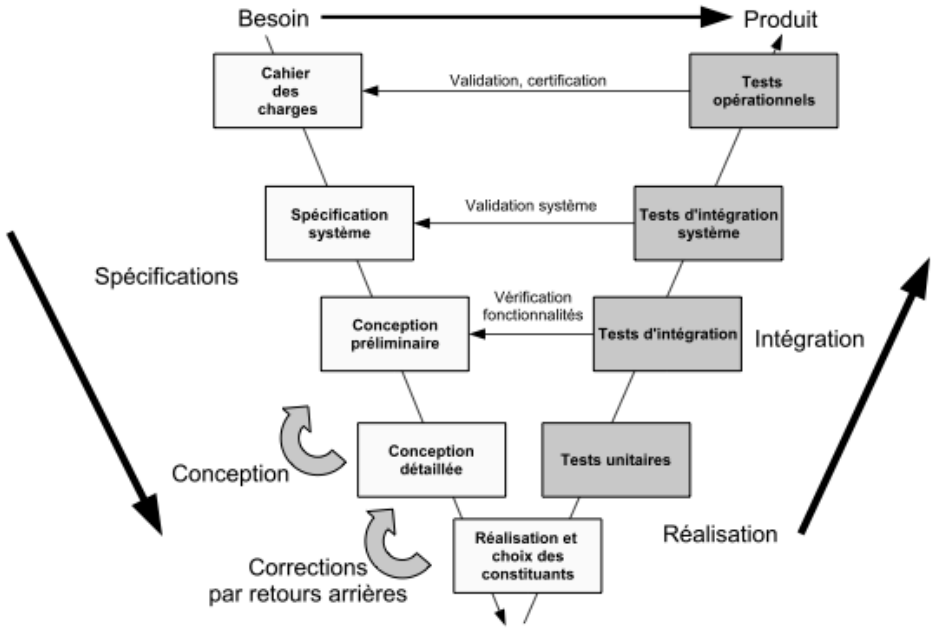

Figure 2.12 : Le modèle de développement en V et l'analyse formelle [Broy & Slotosch 1999]

lors de la première phase d'implémentation. D'un autre côté, des solutions peuvent être apportées par l'utilisation de méthodes formelles, permettant de générer un modèle exécutable pour détecter rapidement des erreurs de conception.

En outre, la prise en compte du contexte n'est pas toujours systématique lors de la proposition d'une solution de vérification formelle en contexte industriel. En effet, la plupart des approches de vérification proposées sont appliquées sur l'automate du système global, afin de vérifier des propriétés sur le modèle du système. Ce n'est qu'a posteriori, que les techniques classiques de réduction sont appliquées afin de réduire l'occurrence de l'explosion combinatoire inhérente au *model checking*. Dans ce mémoire, nous adoptons une approche orthogonale en appliquant la méthode de réduction non plus sur le système global, mais plutôt sur la spécification du système, sur son l'environnement ainsi que sur les propriétés à valider.

En effet, malgré le fait que la technique de vérification par contrôle de modèles soient élégante et automatique, elle souffre de plusieurs contraintes qui limite son large usage en milieu industriel. En effet, la puissance de la vérification formelle s'accompagne de la nécessité d'une forte connaissance de la logique pour en faire bon usage. Une des solutions pour remédier à ce problème est de spécifier le système en se basant sur un modèle plus intuitif pour les utilisateurs (diagrammes UML, langage naturel contrôlé...) et de le traduire ensuite en modèle formel par transformation de modèle. La spécification résultant de cette traduction peut alors être exploitée pour poursuivre un raisonnement rigoureux sur le même système. L'objectif est qu'à la fin du processus de développement formel le système puisse être validé par rapports aux exigences formulées.

2.5 Discussion et synthèse

Nous avons vu dans ce chapitre que la définition d'une spécification claire des entités du domaine est une étape importante pour garantir la cohérence entre les modèles d'analyse et ceux de la conception. Puis nous avons présenté, à travers l'étude d'une approche classique de conception et d'analyse orientée objet, la manière avec laquelle une telle spécification du domaine peut être construite le long du processus de développement. Dans le chapitre 4, nous allons mettre en pratique ces constatations afin de présenter un DSL permettant de recenser ces entités afin de pouvoir les référencer dans les modèles de contextes et des exigences.

Tout comme pour la spécification du domaine, nous avons détaillé l'intérêt de formaliser l'information sur le contexte d'un système logiciel embarqué interagissant avec son environnement. Ceci est particulièrement pertinent dans le cadre de notre approche de vérification formelle dans la mesure où il va nous permettre d'éviter le problème de l'explosion combinatoire inhérent aux modèles de taille industrielle. Nous avons aussi montré la formalisation de la composition des modèles de contextes et celui du système à valider présentée par les travaux précédents. D'autres travaux ont proposé des formalismes afin de bien capturer et formaliser ces interactions au cours d'une approche de vérification formelle. Nous présentons dans le chapitre 5 un langage de spécification et formalisation du contexte. La particularité de ce dernier est qu'il se base sur les cas d'utilisation UML, largement adopté en industrie, en les étendant avec des scénarios d'interactions. Ces scénarios sont exprimés par une composition d'étapes simples exprimées avec du langage naturel contrôlé afin de rester proche des artefacts manipulés au cours des processus de développement industriel, toujours dans un souci de favoriser son adoption.

Nous avons aussi présenté le domaine de l'ingénierie des exigences et ont insisté sur la difficulté d'obtenir une spécification formelle des exigences pour la vérification formelle. Les travaux que nous avons présentés sur l'utilisation du langage naturel et des patrons de spécification ont démontré leur efficacité pour contribuer à résoudre cette problématique. Pour cette raison, nous avons décidé de combiner ces deux axes de recherche sur les exigences afin de définir un langage de spécification des exigences pour combler l'écart entre les exigences présentées dans les cahiers des charges et les propriétés formalisées. Ce dernier utilisera la spécification du domaine ainsi que les patrons de spécification de propriétés tout en prenant en compte les liens avec les modèles de contextes.

Dans la deuxième partie de ce mémoire, nous présentons ces langages et montrons la manière avec laquelle nos travaux contribuent à intégrer les activités de vérification au sein des processus de développement industriels. Mais avant d'en arriver là, nous présentons, dans le chapitre suivant, le cas d'étude utilisé le long de ce mémoire pour illustrer les concepts proposés ainsi que le langage CDL et la chaîne d'outils OBP.

3 La méthodologie CDL et l'outillage OBP

Suite à l'introduction du langage CDL et de la chaîne d'outils OBP dans le premier chapitre, nous détaillons dans celui-ci le principe de la vérification à l'aide du langage CDL. Nous expliquons la prise en compte de l'environnement des modèles à valider ainsi que la spécification des propriétés à vérifiées. En suite, nous détaillons la manière avec laquelle cette approche contourne le problème de l'explosion combinatoire, inhérent aux techniques de vérification par *Model Checking*. Toutefois, dans le but d'illustrer ces concepts, nous introduisons notre étude de cas dans la première section de ce chapitre. Celle-ci nous servira de fil conducteur dans ce mémoire.

3.1 Présentation du cas d'étude

3.1.1 Le système contrôleur SM

Le cas d'étude retenu pour illustrer puis valider notre approche est un sous-système *SM* (*System Manager*) d'un système militaire. Le logiciel embarqué de ce système est responsable de la commande et du contrôle de l'ensemble du système, ses modes internes et ses actions en réponse aux informations reçues de l'environnement. Durant son fonctionnement, le système *SM* est relié à des périphériques (radars, capteurs, rampes de lancement...) et il gère la connexion des opérateurs au système via des HMIs (*Human Machine Interface*) (figure 3.1).

Le système étudié est critique, soumis à des contraintes de fiabilité et de sureté strictes. SM a été modélisé en UML2 à l'aide de l'outil Rhapsody puis implémenté en code Java. Le modèle comportemental du système a été traduit vers le langage Fiacre [Berthomieu *et al.* 2007, Berthomieu *et al.* 2008]. Les scénarios d'utilisation du *SM* ont été formalisés en CDL. Les exigences exprimées dans le cahier des charges du composant *SM* sont au nombre de 70.

3.1.2 Architecture du système

Le *SM* permet aux opérateurs de lancer une mission, d'arrêter une mission ou d'éteindre le système. Pour ce faire, les opérateurs peuvent, via des consoles appelées HMI, envoyer des requêtes. Les HMIs envoient des requêtes au SM au moyen de communications asynchrones. Le SM répond par l'affirmative ou la négative et renseigne au besoin les HMIs de l'évolution du système. Dans certains cas, le SM doit communiquer avec d'autres

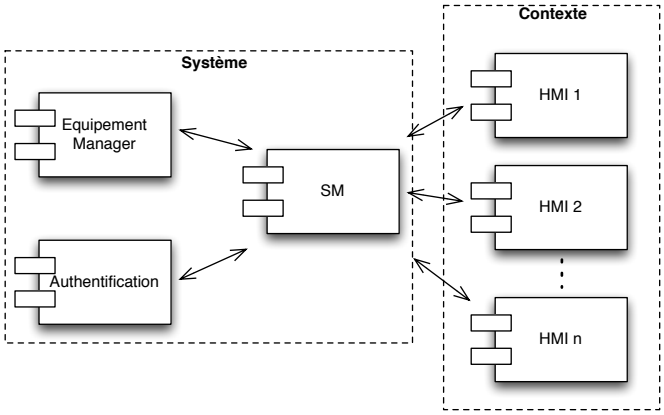

Figure 3.1 : Le composant SM au sein de son environnement

acteurs, comme l'authentification ($Authentification$) ou le gestionnaire d'équipements ($EquipmentManager$), pour décider s'il accède ou non à une requête. Pour la construction des contextes, les acteurs *HMIs* sont considérés comme des acteurs inclus dans le contexte et le gestionnaire d'équipements et l'authentification sont considérés comme des processus du système (figure 3.1).

3.1.3 Cas d'utilisation

Durant l'initialisation, le SM commence par demander au gestionnaire d'équipements quelles sont les consoles atteignables. Une fois renseigné, il informe ces dernières qu'il est prêt à recevoir des requêtes de *login*. Pour pouvoir décider s'il accède à la requête de *login*, le SM consulte d'abord l'authentification. Si les informations sont correctes, il informe, la console correspondante de son état courant en plus d'accepter la commande. La figure 3.2 illustre un exemple d'interaction entre le SM et son environnement.

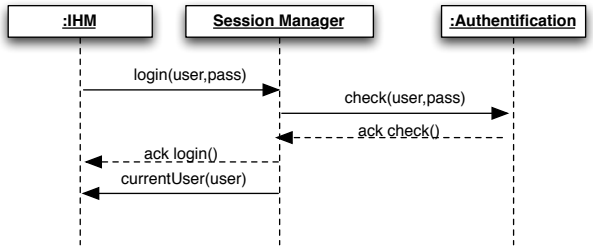

Figure 3.2 : Exemple de scénarios d'interaction du SM.

3.1.4 Exemple d'exigence à vérifier sur le modèle

Pour compléter l'illustration du cas d'étude, nous présentons une exigence R extraite du cahier de charge du système SM. Ainsi, l'approche présentée dans ce mémoire a pour

but de faire le lien entre la forme informelle des exigences présentées dans le cahier des charges et les propriétés formalisées dans les modèles CDL. Nous allons détailler par la suite le processus permettant la mise en oeuvre de ce lien.

Exigence *R* : *« During the initialization phase, registered console might ask the SM for logging. The SM have to associate an identifier to asking console within the $MaxD_{log}$ delay. »*

Ainsi, au cours de la procédure d'initialisation, *SM* doit associer un identificateur à chaque périphérique (*HMI*) du système ayant demandé à se *loguer*, avant un délai $MaxD\_log$. Chaque périphérique *logué* peut demander une opération et reçoit en retour un rôle avant un délai $MaxD\_oper$. L'initialisation s'achève avec succès, lorsque *SM* a affecté tous les identificateurs et les rôles aux périphériques. Dans le chapitre 9, nous présentons l'application de l'approche proposée dans ce mémoire pour la génération d'une spécification formelle des exigences textuelles telle que l'exigence R.

3.2 Les principes mis en œuvre de la vérification avec CDL

3.2.1 Prise en compte de l'environnement des modèles à valider

Un système réactif communique de façon continue avec son environnement de manière à répondre à ses sollicitations. Pour appliquer une vérification par model-checking, il est donc nécessaire de fermer ce système en décrivant son environnement. De façon générale, dans le domaine de l'embarqué, les environnements ne sont pas quelconques, mais sont bien identifiés. Ils sont définis par les connaissances que l'ingénieur a des différentes phases qui caractérisent le cycle de vie du système à concevoir. De plus, les exigences incluses dans la spécification sont, dans la majorité des cas, relatives à ces phases, qui régissent le comportement du système, tel que les phases d'initialisation, d'échange de données, de changement de modes, de reconfiguration, ou encore, les modes dégradés. Ces modes sont activés par des événements issus de l'environnement du système. Ce dernier est composé d'autres entités logicielles ou sous-systèmes. Il devient donc alors inutile de prendre en compte la totalité des comportements de l'environnement pour la vérification de chaque exigence. De ce fait, l'idée consiste à se focaliser sur une partie pertinente du comportement du système et sur les propriétés que l'on souhaite vérifier. Le fait de restreindre les comportements de l'environnement permet de réduire la taille de l'automate global obtenu après composition (synchronisation forte) du système avec ce dernier.

Sur cette idée, une méthode de description de l'environnement sous la forme de cas d'utilisation (ou contextes) a été développée [Roger 2006, Dhaussy & Boniol 2007, Dhaussy *et al.* 2009, Dhaussy *et al.* 2011]. Elle propose à l'utilisateur de spécifier, dans le langage CDL[1], le comportement souhaité de l'environnement du modèle à valider. Le langage CDL, présenté en détail dans la section 3.3, est intégré dans une méthodologie de vérification et exploité par l'outillage OBP.

[1]Pour la syntaxe détaillée, voir [Dhaussy & Roger 2011] disponible sur le site http://www.obpcdl.org

3.2.2 Identification des contextes pour contourner l'explosion combinatoire

Lors de la vérification de propriétés sur un modèle, un model-checker explore tous ses comportements et vérifie, lors de l'exploration, s'il y a violation des propriétés exprimées. Toutefois, il est souvent le cas que le nombre de comportements atteignables du modèle exploré est beaucoup trop grand à cause de l'explosion du nombre d'états du système. Lors de la mise en œuvre du model-checking, traditionnellement, la description des contextes, c'est-à-dire des interactions entre l'environnement et le modèle, est incluse dans la description du modèle du système (Figure 3.3.a) à valider. Le contexte prend la forme d'un automate composé avec d'autres automates du modèle à valider lors de l'exploration. A ce niveau, pour éviter l'explosion combinatoire, il faut identifier et expliciter des contextes qui soient suffisamment restrictifs pour limiter le nombre de comportements à explorer lors du model-checking. C'est le premier stade d'une politique de "diviser pour régner" en activant des sous-ensembles des comportements du modèle. Mais l'expérience montre que pour éviter l'explosion, les contextes doivent décrire un nombre limité de comportements de l'environnement [Dhaussy *et al.* 2011]. Ceci implique donc la description d'une multitude de contextes ce qui peut être très pénalisant dans un processus industriel.

Dans l'approche développée dans [Dhaussy *et al.* 2011], il a été choisi d'expliciter les contextes séparément du modèle, dans un formalisme spécifique qui permet de les traiter avant la composition avec le modèle. Pour identifier les contextes, l'utilisateur se base sur la connaissance qu'il a de l'environnement du système ou de ses parties, pour les spécifier formellement. Ils correspondent aux modes d'utilisation du composant modélisé. Dans le contexte des systèmes embarqués réactifs, l'environnement de chaque composant d'un système est souvent bien connu. Il est donc plus simple d'identifier cet environnement que de chercher à réduire l'espace des configurations du modèle du système à explorer.

L'objectif est de disposer (Figure 3.3.b) de la description des sous-ensembles des comportements des acteurs de l'environnement (Contexte$_i$, i $\in$ [1..n] en Figure 3.3.b) et des propriétés (Propriété$_i$) associées à ces comportements. Cette identification des contextes peut déjà permettre de contourner l'explosion lors de l'exploration du modèle.

Pour la mise en œuvre de cette approche, le processus de développement du système doit inclure une étape de spécification de l'environnement permettant d'identifier explicitement des ensembles de comportements. L'hypothèse forte qui est faite ici pour mettre en œuvre ce processus méthodologique est que le concepteur est capable d'identifier les interactions entre le système et son environnement. Chaque contexte exprimé au départ est considéré comme fini, c'est-à-dire que les scénarios décrits par ce contexte ne présentent pas de comportements itératifs infinis. Cette hypothèse est justifiée, en particulier dans le domaine de l'embarqué, par le fait que le concepteur d'un composant logiciel doit connaître précisément et complètement le périmètre (contraintes, conditions) de son utilisation pour pouvoir le développer correctement. Il serait nécessaire d'étudier formellement la validité de cette hypothèse de travail en fonction des applications ciblées.

Au delà de l'identification et l'exploitation d'un ensemble de contextes, l'outil OBP met en œuvre une technique complémentaire de réduction basée sur un partition-

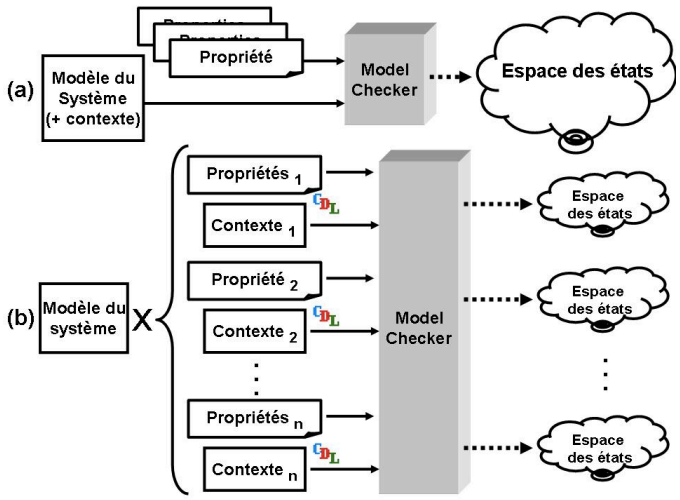

Figure 3.3 : Exploration sans (a) et avec (b) identification séparée des contextes.

nement automatique de chaque contexte identifié. Cette technique est décrite dans [Dhaussy *et al.* 2011] et rappelée en section 3.4.2.

3.2.3 Spécification des propriétés

Nous avons évoqué précédemment la difficulté pour un utilisateur d'exprimer des exigences dans des formalismes à base de logique temporelle. Pour faciliter cette expression, CDL intègre des patrons de définition qui permettent de spécifier des propriétés comportementales et temporelles. De nombreux auteurs [Dwyer *et al.* 1999, Smith *et al.* 2002, Konrad & Cheng 2005a] ont proposé différentes formes de patrons de définition de propriétés pour formuler les propriétés. Un patron est une structure syntaxique textuelle qui permet un mode d'expression contraint mais plus proche des langages que les ingénieurs ont l'habitude de manipuler.

Une autre voie de simplification de l'expression des exigences provient du fait que, dans les documents d'exigences, celles-ci sont souvent exprimées dans un contexte donné de l'exécution du système. Les exigences sont associées à des phases spécifiques d'utilisation du système. Elles sont souvent relatives à des scénarios particuliers (initialisation, mode nominal, mode dégradé, etc.). Il n'est donc pas nécessaire pour une exigence donnée de prendre en compte tous les comportements possibles de l'environnement. Il suffit de considérer le sous-ensemble des comportements concernés par cette exigence.

Dans les travaux décrits dans [Konrad & Cheng 2005a], les auteurs ont proposé d'identifier la portée (appelée *Scope*) d'une propriété en permettant à l'utilisateur de préciser le contexte temporel d'exécution de la propriété à l'aide d'opérateurs (*Global*, $Before$, $After$, *Between*, $After-Until$). Ceux-ci permettent de localiser les exigences à vérifier dans un contexte temporel particulier d'exécution du modèle à valider. Ce *Scope* indique si la propriété doit être prise en compte, par exemple, au cours de tous les cas d'utilisation du modèle, avant, après ou entre des occurrences d'évènements. Les travaux

sur CDL se sont inspirés de cette notion lors de la mise en œuvre de l'outillage OBP.

À partir des exigences fournies, des propriétés sont formalisées et un ou des modèles du comportement de l'environnement sont construits. Cette formalisation est réalisée à partir d'une interprétation d'exigences textuelles et d'une description de l'environnement du système modélisé. La rédaction d'exigences formelles et la modélisation du contexte sont d'autant plus aisées que, d'une part, la description du comportement de l'environnement est déjà formalisée (cas d'utilisation, scénarios, diagramme de séquence, ...) et que, d'autre part, les exigences sont exprimées sans ambiguïté.

3.2.4 Les bénéfices de l'approche

Cette approche a été expérimentée sur plusieurs cas d'étude industriels avec différents partenaires [Dhaussy *et al.* 2009] dans le cadre du projet TOPCASED [Farail *et al.* 2006]. L'apport des modèles CDL est d'offrir un cadre formel pour décrire les interactions entre des acteurs de l'environnement et le modèle, ainsi que les exigences. Compte tenu du type de cas traités (domaine des protocoles), une part importante des exigences a pu être exprimée avec les patrons proposés. L'approche a permis aux utilisateurs de formaliser le comportement de l'environnement de leur système et de préciser, dans un ensemble de modèles CDL, les cas d'utilisation du système ou composant développé. Les ingénieurs se sont appropriés le langage et y ont trouvé un intérêt pour mieux rédiger leurs spécifications.

Les modèles de conception traités étaient de taille suffisamment grande pour montrer l'intérêt de l'exploitation des modèles CDL pour contourner l'explosion combinatoire. L'identification des contextes CDL et la technique de partitionnement ont permis de restreindre l'exécution des modèles et de mener complètement les explorations. Les patrons de propriétés proposés dans CDL simplifient l'écriture des exigences dans la limite où leur expression est possible avec ces patrons. Les résultats obtenus dans les expérimentations sont encourageants car ils contribuent à une meilleure appropriation industrielle des techniques de vérification formelle [Dhaussy *et al.* 2009]. Mais cette approche peut poser aussi certaines difficultés, que nous abordons dans la section suivante, et qui ouvrent un ensemble d'axes de travaux de recherche à entreprendre.

3.3 Le langage CDL

Ce DSL s'inspire des Use Case Chart de [Whittle 2006] basé sur des diagrammes d'activités et de séquences. Ce formalisme permet de décrire plusieurs entités (nommées *acteurs*) contribuant à l'environnement et pouvant s'exécuter en parallèle, comme les acteurs HMI de la figure 3.1. Doté d'une syntaxe graphique et textuelle, un modèle CDL décrit, d'une part, un scénario avec des diagrammes d'activités et de séquences. D'autre part, il décrit les propriétés à vérifier en se basant sur des patrons de définition de propriétés. Un métamodèle de CDL a été défini et une sémantique décrite en terme de traces [Dhaussy *et al.* 2011], s'inspirant des travaux de [Haugen *et al.* 2005] et [Whittle 2006].

3.3.1 Description hiérarchique du contexte

Un modèle CDL est structuré en trois niveaux. Au premier niveau, des diagrammes de cas d'utilisation décrivent, par des diagrammes d'activités et de manière hiérarchique, les enchaînements d'activités des entités s'exécutant en parallèle et constituant l'environnement. Les diagrammes de ce niveau font référence à des diagrammes du même niveau ou des scénarios décrits au niveau 2 également sous la forme de diagrammes d'activités. Les diagrammes de niveau 2 décrivent des enchaînements de scénarios, ceux-ci étant décrits au niveau 3 par des diagrammes de séquences UML2.0 [ITU 1996] simplifiés.

Lors de la compilation d'un modèle CDL, les diagrammes correspondant à chaque acteur sont dépliés (mis à plat) puis entrelacés pour respecter la sémantique du comportement parallèle des acteurs. Nous pouvons donc considérer que le modèle est structuré par un ensemble de diagrammes de séquences (MSCs), reliés entre eux à l'aide de trois opérateurs: celui de la séquence (*seq*), l'opérateur parallèle (*par*) et l'alternative (*alt*). L'entrelacement d'un contexte, décrit par un ensemble de MSCs, génère un graphe représentant toutes les exécutions des acteurs de l'environnement considéré. Ce graphe est ensuite partitionné, de manière à générer un ensemble de sous graphes correspondants aux sous-contextes, comme mentionné en 3.4.2. Lors de l'exploration par le vérificateur, chaque sous-graphe est composé (Figure 3.6) avec le modèle à valider et les propriétés sont vérifiées sur le résultat de cette composition.

CDL permet aussi de rattacher des propriétés sur des MSCs spécifiques de manière à simplifier la formalisation des propriétés. Dans cet article, les contextes sont modélisés à l'aide d'une sous-partie du langage CDL suffisante ici pour la description de l'environnement du système présenté. En particulier, les compteurs ne sont pas utilisés. La figure 3.4 illustre graphiquement et partiellement un modèle du contexte du système S_CP. Les cas d'utilisation illustrés Figure 3.2 sont modélisés et complétés pour créer le modèle de contexte.

Dans les cas d'applications industrielles, compte tenu de la complexité potentielle des interactions entre le modèle et son environnement, la construction d'un seul modèle CDL peut être difficile. Il est donc réaliste de considérer que l'utilisateur spécifie un ensemble de modèles CDL, chacun correspondant à des cas d'utilisation particuliers du modèle à valider.

3.3.2 Syntaxe formelle de CDL

Nous considérons qu'un contexte est décrit sous la forme d'une composition séquentielle, parallèle ou alternative, de diagrammes de séquences. Le langage CDL possède trois opérateurs simples à manipuler, *par*, *alt* et *seq*, respectivement notés par la suite $\|$, $+$ et ;, et qui permettent de structurer l'expression d'un scénario par leur combinaison.

Formellement, un contexte est un système fini produisant et recevant des évènements décrits par la grammaire suivante:

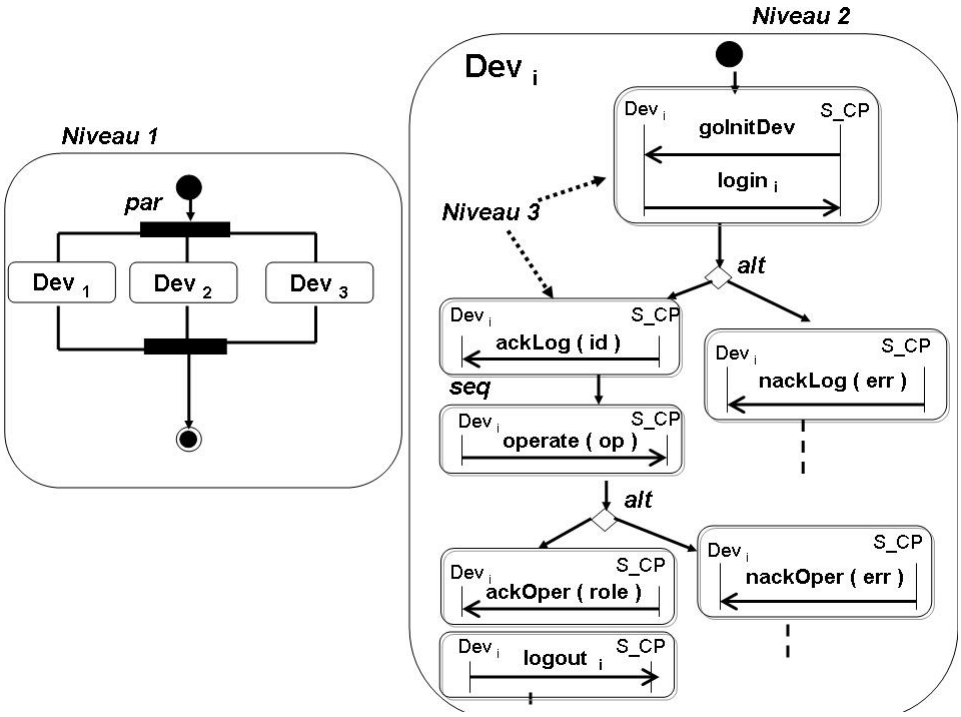

Figure 3.4 : Illustration d'un modèle CDL partiel du contexte du système SM

$$\begin{array}{lcl} C & ::= & M \mid C_1; C_2 \mid C_1 + C_2 \mid C_1 \| C_2 \\ M & ::= & \mathbf{0} \mid a!; M \mid a?; M \text{ with } a \neq null_\sigma \end{array}$$

Un contexte est soit un simple MSC M composé d'une séquence d'évènements d'émission $a!$ et de réception $a?$ terminée par un MSC terminal qui ne fait plus rien ($null_c$), soit une composition séquentielle de deux contextes ($C_1; C_2$), soit une alternative entre deux contextes ($C_1 + C_2$), soit, enfin, une composition parallèle de deux contextes ($C_1 \| C_2$).

Par exemple, considérons la figure 3.4. Nous considérons ici que l'environnement est composé de 3 acteurs, Dev_1, Dev_2 et Dev_3, s'exécutant en parallèle. Le modèle peut être formalisé de la façon suivante [2] :

$$\begin{array}{lcl} C & = & Dev_1 \parallel Dev_2 \parallel Dev_2 \\ Dev_i & = & Log_i;\ (Oper\ +\ (\ nackLog\ (err)\ ?\ ;\ \ldots\ ;\ null_c)) \\ Log_i & = & (goInitDev\ ?\ ;\ login_i\ !) \\ Oper & = & (\ ackLog\ (id)\ ?\ ;\ operate\ (op)\ !\ ; \\ & & \qquad (\ Ack_i\ +\ (\ nackOper\ (err)\ ?\ ;\ \ldots\ ;null_c))) \\ Ack_i & = & (\ ackOper\ (role)\ ?\ ;\ logout_i\ !\ ;\ \ldots;\ null_c) \\ Dev_1, & & Dev_2,\ Dev_3\ =\ Dev_i \quad avec\ i\ =\ 1,\ 2,\ 3. \end{array}$$

[2] Pour l'illustration dans ce papier, nous considérons ici que les comportements des acteurs se prolongent, ce qui est noté par les pointillés « ... ».

3.4 L'OUTILLAGE OBP

3.4.1 Transformation des modèles CDL

Pour mener les expérimentations, l'outil prototype OBP[3] a été mis en œuvre sur plusieurs cas d'étude industriels [Dhaussy *et al.* 2009]. Comme illustré sur la figure 3.5, OBP traduit les modèles CDL en programme Fiacre [Farail *et al.* 2008]. En effet, à partir d'un modèle CDL, OBP génère un ensemble de graphes acycliques de contexte. Actuellement, chaque graphe est transformé en un automate Fiacre assimilable par les explorateurs TINA [Berthomieu *et al.* 2004] ou *OBP Explorer*. Ceux-ci représentent l'ensemble des interactions possibles entre le modèle et l'environnement. Pour valider le modèle, il est nécessaire de composer chaque graphe avec le modèle. Chaque propriété référencée dans le modèle CDL doit être vérifiée sur le résultat de cette composition.

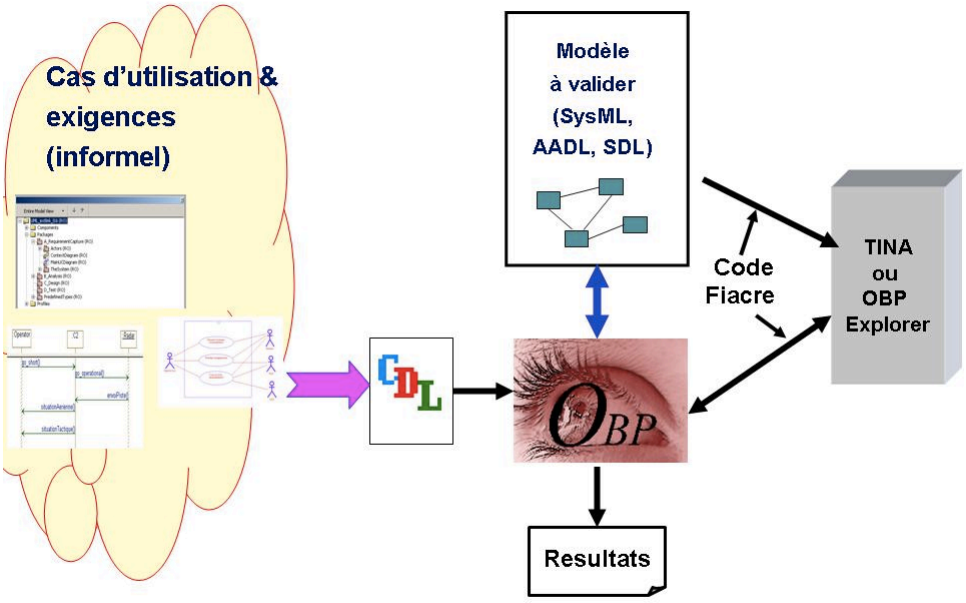

Figure 3.5 : Transformation de modèle CDL avec OBP

OBP génère, pour chaque propriété, soit un automate observateur pour OBP Explorer, soit une formule au format SELT [Berthomieu *et al.* 2004] pour l'outil TINA. Compte tenu de l'expressivité des patrons de propriétés, la génération de formules logiques SELT est limitée aux propriétés pouvant être converties en logique SELT. Dans le cas d'*OBP Explorer*, une analyse d'accessibilité est menée sur le résultat de la composition entre un graphe généré, un ensemble d'observateurs et le modèle. Lorsque la restriction des comportements du modèle, décrite précédemment, n'est pas suffisante, lors de l'exploration, un

[3] OBP_t (OBP for TINA) accessible sur http://www.obpcdl.org

second levier de réduction de l'espace des états est mis en œuvre par un partitionnement des contextes.

3.4.2 Partitionnement automatique des graphes de contexte

Si, pour un graphe de contexte donné, la composition bute sur l'explosion combinatoire, l'analyse devient impossible. Chaque contexte est alors traité séparément et partitionné automatiquement en un ensemble de scénarios qui sont nommés (*souscontextes*) plus réduits (Figure 3.6). Ceci est rendu possible du fait que les contextes sont préalablement spécifiés séparément du modèle. Un algorithme de partitionnement peut donc leur être appliqué aisément. Ces sous-contextes prennent la forme d'automates qui sont chacun composés avec le modèle. C'est le deuxième stade de la politique "diviser pour régner".

C'est cette partition du contexte en un ensemble de graphes qui permet d'aboutir, lors de la composition, à des systèmes de transitions de taille limitée rendant possible l'analyse d'accessibilité et le model-checking.

Les processus de vérification suivants sont alors équivalents : (i) composer le contexte (Contexte$_i$) et le modèle puis vérifier alors les propriétés (Propriétés$_i$) sur la composition des deux, (ii) partitionner le contexte en K sous contextes (scénarios) puis composer successivement chaque scénario avec le modèle et vérifier les propriétés sur le résultat de chaque composition. L'algorithme récursif de partitionnement mis en œuvre dans l'outil OBP est décrit dans [Dhaussy *et al.* 2011].

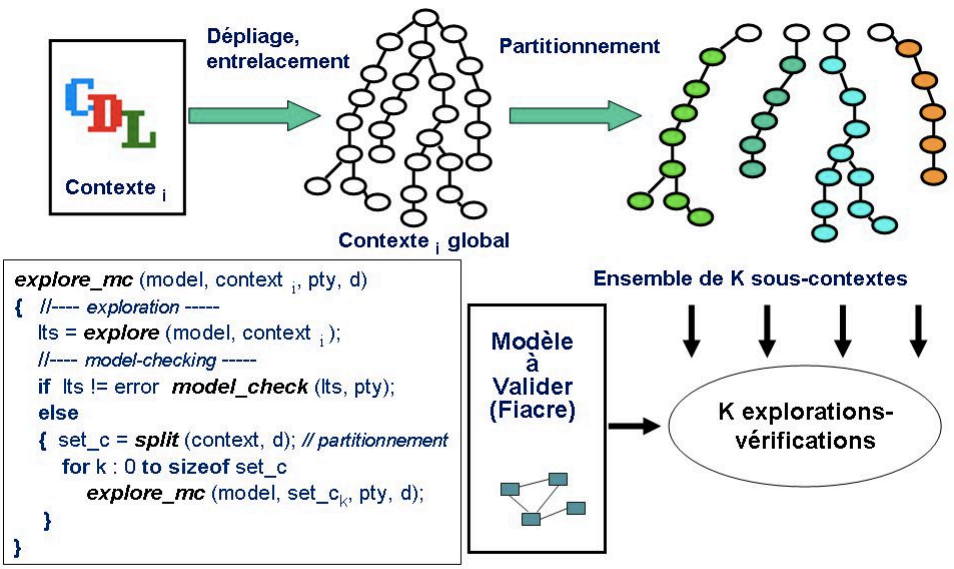

Figure 3.6 : Partitionnement d'un contexte et vérification pour chaque partition

De fait, une vérification globale est transformée par plusieurs vérifications plus petites.

Il faut noter que la technique de partitionnement mise en œuvre respecte le principe suivant : pour un contexte donné, l'union des exécutions, décrite par l'ensemble des sous-contextes générés par le partitionnement du contexte, inclut les exécutions décrites par ce contexte initial. Les propriétés sont préservées par le partitionnement du contexte comme démontré dans [Roger 2006].

Toutefois, afin d'importer des modèles de plus haut niveau d'abstraction telle que UML, AADL ou SDL, il est nécessaire de développer des traducteurs adéquats comme mentionnés précédemment. Ceci dans le but de générer automatiquement, à partir des modèles de conception de l'utilisateur, des programmes formels (en FIACRE par exemple).

Dans la configuration actuelle, OBP délivre à l'utilisateur un retour de preuve lui indiquant si chaque propriété à vérifier est détectée comme vraie ou fausse. En cas d'échec d'une propriété (détectée comme fausse), OBP indique à l'utilisateur les séquences d'exécution de l'environnement (contre-exemples éventuellement filtrés), concerné durant la preuve. Cette indication peut l'aiguiller sur le scénario ayant mis en échec la propriété. Des travaux en cours de développement visent à obtenir des facilités pour restituer des données de plus haut niveau dans le modèle de l'utilisateur, lui permettant de constituer son diagnostic.

3.5 La méthodologie OBP

3.5.1 Le processus de vérification

Le processus méthodologique de vérification d'exigences proposé suppose l'existence d'un ensemble d'artefacts. Tout d'abord, le modèle de conception sur lequel s'applique la vérification des exigences doit pouvoir être simulable et traduisible dans un format accepté par le vérificateur. Ensuite, les exigences sont disponibles et peuvent être formalisées sous la forme de propriétés logiques. Enfin, les interactions entre l'environnement et le modèle doivent être décrites précisément. Le modèle simulable, les exigences et les interactions avec l'environnement constituent les données pertinentes et suffisantes pour aborder l'approche que nous proposons. Celle-ci inclut donc les phases suivantes (Figure 3.7):

- A partir d'un modèle de conception fourni par l'industriel, un modèle formel est généré (manuellement ou semi automatiquement) par une extraction des données utiles du premier modèle et une traduction adéquate. Ces données permettent d'obtenir un modèle comportemental formel simulable par l'explorateur du vérificateur. Nous supposons ici que le modèle formel généré est sémantiquement conforme à la partie utile du modèle fourni par l'industriel. Cette conformité doit être assurée par la traduction. Nous ne décrivons pas ces transformations de modèle, mais des informations à ce sujet peuvent être trouvées dans des projets tels que TopCased[4] ou Oméga[5]. Dans nos expérimentations, nous avons construit manuellement les modèles au format du vérificateur utilisé[6].

[4] http://www.topcased.org
[5] http://www-Omega.imag.fr
[6] Les premières expérimentations ont été réalisées avec le langage IF2 [Bozga *et al.* 1999] avant d'être remplacé par le langage Fiacre pour des raisons de disponibilité d'outillage.

- A partir d'une interprétation d'exigences textuelles (en langage naturel) et d'autres éléments de spécification disponible fournie dans des documents d'exigences, les propriétés et les scénarios CDL sont formalisés. La rédaction des propriétés formelles et la modélisation du contexte sont d'autant plus aisées que, d'une part, la description du comportement de l'environnement est déjà formalisée (cas d'utilisation, scénarios, diagrammes de séquences, etc.) et que, d'autre part, les exigences sont exprimées sans ambiguïté.

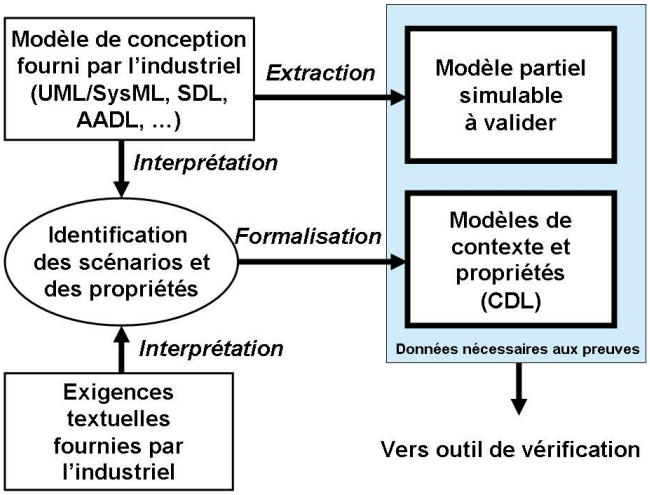

Figure 3.7 : Processus de prise en compte des données d'exigences et du modèle à valider.

A ce niveau, il nous semble important que les exigences et le contexte soient bien dissociés des éléments du modèle de conception et formalisés de façon non ambiguë. Ils doivent également porter sur des éléments bien identifiés du modèle de conception.

Une condition préalable pour la mise en œuvre de ce processus est de disposer, de la part de l'industriel, de spécifications qui permettent d'identifier les contextes et les exigences qui peuvent être soumises à la vérification et qui peuvent être reliées à un contexte. Le processus de développement doit inclure une étape de spécification de l'environnement, permettant d'identifier un ensemble complet de toutes les interactions entre l'environnement et le modèle, ce qui doit assurer un taux de couverture de 100%. L'atteinte de cette couverture correspond à l'hypothèse formulée précédemment, qui postule que le concepteur est capable d'identifier toutes les interactions possibles entre le système et son environnement. L'ensemble des interactions doit lui être fourni formellement comme un résultat du processus d'analyse de l'architecture logicielle conçue. Compte tenu de la complexité de l'ensemble des interactions, il est préférable que l'utilisateur construise un ensemble structuré de modèles CDL spécifiques, chacun correspondant à des cas d'utilisation spécifiques.

En ce qui concerne la formalisation des propriétés, les exigences comportent souvent beaucoup d'informations sur le système et son environnement. Pour simplifier leur formalisation avec les patrons, elles doivent être décomposées en exigences élémentaires. Par

exemple, dans l'étude de cas décrits précédemment, l'exigence R (listing 1) peut être décomposée en trois sous-exigences comme suit:

R1: *"Au cours de la procédure d'initialisation, SM doit associer un identificateur à chaque périphérique (HMI) du système ayant demandé à se loguer, avant un délai MaxD_log."*

R2: *"Chaque périphérique logué peut demander une opération et reçoit en retour un rôle avant un délai MaxD_oper."*

R3: *"L'initialisation s'achève avec succès, lorsque SM a affecté tous les identificateurs et les rôles aux périphériques."*

Une fois ce travail de décomposition des exigences réalisé, la formalisation avec les patrons CDL est facilitée. Mais malgré cette décomposition, certaines exigences peuvent nécessiter un dialogue avec la personne qui a spécifié l'exigence initiale. Par exemple, l'exigence $R3$ mérite une demande de clarification concernant la signification de la mention "avec succès". Les méthodologies étudiées dans le domaine de l'ingénierie des exigences par exemple dans [Cheng 2007, Lamsweerde 2009, Yue *et al.* 2010] sont une source d'inspiration pour compléter ce travail. Compte tenu de l'ampleur du sujet, nous choisissons de ne pas aborder ces aspects dans cette étude.

3.6 Discussion et synthèse

Au cours d'un processus de développement de systèmes embarqués, les activités de vérification peuvent être vues comme une aide à la mise au point, beaucoup plus puissante que la simple simulation, des modèles avant la génération du code embarqué. Pour cette raison, dans le cadre de cette thèse, nous cherchons à construire une méthodologie permettant d'améliorer les activités de vérification de logiciels dans un contexte industriel. Nous souhaitons construire des modèles fiables permettant, d'une part, la génération de code et, d'autre part, la vérification formelle d'exigences sur ces modèles.

Toutefois, le processus de validation formelle d'un modèle implique la description formelle de son environnement et des exigences sur le comportement qui lui sont associées. De ce fait, on peut distinguer donc deux phases importantes.

Phase I: La formalisation des exigences et des interactions entre le modèle à valider et son environnement,

Phase II: Les preuves des exigences avec un outil de vérification.

La phase I a pour but de formaliser le comportement du modèle dans son environnement avec la vue souhaitée par l'industriel (liste des acteurs impliqués, description des interactions, formalisation des exigences...). Ce dernier utilise au cours de cette phase des langages de modélisation de haut niveau afin de monter en niveau d'abstraction et pouvoir gérer la complexité des modèles industriels. Néanmoins, il doit construire ses modèles en y apportant les données nécessaires à une formalisation permettant de lever toutes les ambigüités d'interprétation, d'une part, sur le contexte d'emploi de son modèle et, d'autre part, sur les exigences. Ainsi, nous avons discuter lors de ce chapitre du rôle important que peut jouer les langages intermédiaires tels que CDL dans la description de ces scénarios d'interactions et des exigences. Toutefois, l'écart sémantique entre les modèles que cet industriel a l'habitude de manipuler et les modèles formels demeure un

frein à leur adoption, car l'effort nécessaire à la production de ces artefacts formels est considérable. C'est sur ce constat que nous avons proposé un ensemble de langage, dits orientés utilisateur, afin de combler cet écart et automatiser la génération des modèles formels (modèles CDL dans notre cas).

La phase II ne s'intéresse qu'à la partie technique de la vérification . En effet, elle se base sur les modèles produits lors de la phase I, et met en œuvre des formalismes qui pourraient être, à terme, cachés à l'utilisateur. Jusqu'à présent, cette phase implique de formaliser, avec le langage CDL, le comportement de chaque acteur de l'environnement et d'exprimer les exigences par rapport à cette formalisation. Dans la partie qui suit, nous proposons les formalismes nécessaires, permettant la spécification et la formalisation des exigences et des modèles de contextes ainsi que la génération automatique des modèles et propriétés CDL. De ce fait, la formalisation effectuée durant la phase I permettra la génération automatique des modèles nécessaires à la phase II.

FORMALISMES PROPOSÉS

TELECOM Bretagne

4 Spécification du domaine

Au cours de la première partie de ce mémoire, nous avons présenté un état de l'art concernant l'exploitation des techniques de vérification formelles dans un contexte industriel. Nous avons pu dégager des points précis limitants l'utilisation systématique de ces techniques pour améliorer la fiabilité des logiciels développés. L'interprétation des modèles manipulés par les ingénieurs pour en extraire une description formelle précise des spécifications demeure une activité délicate et difficile à mettre en oeuvre. Cela est dû en grande partie à la nature hétérogène et informelle des spécifications textuelles produites lors des premières phases du processus de développement [Richards 2003]. Dans ce chapitre nous présentons un langage de spécification du domaine proposé dans le cadre de la formalisation des exigences et des contextes du système étudié. Notre approche consiste à séparer la définition du vocabulaire du domaine du reste de la spécification des exigences afin de facilité la production d'exigences claires et mieux contrôler le vocabulaire utilisé. En effet, la plupart des incohérences relevées dans les exigences sont causées par des définitions ambiguës, parfois contradictoires, des termes employés. Afin d'éliminer cette source d'incohérence des exigences, nous avons proposé d'utiliser un référentiel des termes (vocabulaire) employés dans les exigences. Les termes définis dans ce vocabulaire serons utiliser pour la spécification des exigences mais aussi pour les modèles de contextes.

4.1 Construction d'une terminologie

Dans cette section nous présentons notre langage DSpec permettant de faciliter la capture et la formalisation des entités du domaine ainsi que leurs relations. La description d'une spécification du domaine à l'aide de modèles DSpec permet de faciliter la prise en compte des entités et concepts liés au domaine dès les premières étapes de la méthodologie. En effet, lors de la spécification des exigences et des scénarios d'interaction entre le modèle à valider et le contexte, le concepteur peut se réfère au vocabulaire du domaine qui liste l'ensemble des entités du domaine (objets, messages, arguments, variables...). Toutefois, en comparaison avec l'approche présentée dans la section 2.1.1 page 13, notre approche propose un certain nombre d'améliorations. Nous proposons l'utilisation de modèles d'entités du domaine pour le travail de conceptualisation au lieur de travailler directement avec des diagrammes de classes UML. Dans un diagramme d'entités du domaine, les éléments du domaine contiennent des *phrases* qui référence ces entités. Notre langage décrit ces entités et permet de faire le lien entre le vocabulaire et les exigences comportementales.

La création d'une terminologie précise est une étape cruciale pour la définition d'une spécification cohérente d'exigences. Généralement, seuls les noms des entités du domaine sont incluses dans les dictionnaires de vocabulaire pour être tracées dans les exigences. Les exigences dans le langage URM sont représentées par le "modèle Sujet-Prédicat", où le sujet ainsi que les prédicats sont définis à l'aide d'un vocabulaire cohérent. Ce vocabulaire contiens les notions qui sont liées dans des phrases elles-mêmes sont regroupées en fonction de leurs sujets. Les phrases peuvent inclurent les verbes, les adjectifs ou les propositions. Ces derniers sont utilisés pour former des phrases de types simples, modales ou conditionnelles.

Le langage DSpec a été proposé pour spécifier les entités et les concepts d'un domaine d'application particulier. Ces entités seront référencées par les exigences et les modèles de contextes. L'idée de référencer les entités du domaine depuis les exigences n'est pas une idée innovante en soit. Elle a déjà été introduite par Kaindl dans [Kaindl 1997].

4.1.1 Spécification du domaine

Le but du modèle conceptuel du domaine est de capturer les entités qui forment le domaine du système étudié. D'après [Wolter *et al.* 2008], une spécification du domaine peut être vue comme l'ensemble des entités métier et les entités du système (logicielles et matérielles). Dans la suite de cette section, nous allons détailler ces deux groupes d'entités que nous proposons par la suite de capturer à l'aide de notre langage DSpec.

4.1.1.1 Les entités métier

Pour construire un modèle du domaine, il faut étudier et identifier les différentes entités du domaine métier. Les sources possibles de ces entités sont:

1. *Les ressources métiers:* qui regroupent l'ensemble des entités, physiques et logiques, qui existent à l'intérieur de l'environnement et des ressources métier. Cela inclus les personnes, l'information, les différents systèmes et les produits qui participent dans le processus métier. Le but de former une base de connaissances à propos de ces entités réside dans le fait qu'elles vont servir par la suite à effectuer des analyses sur l'architecture du système, tracer les changements du domaine et du système et évaluer comment le système répond aux besoins courants du métier. Un exemple de ressources métier pour le système *AFS* sera le terminal permettant l'affichage des messages du *System Manager* aux utilisateurs.

2. *Les processus métier:* Un système à concevoir peut participer dans plusieurs processus métier pour aider à la réalisation de plusieurs objectifs métier. Les cas d'utilisation décrivent des sous-processus qui font partie d'un processus métier plus important qui est automatisé par l'application du système. De ce fait, il est important de comprendre le processus métier puisqu'il est lié à ces cas d'utilisation, qui à leur tour, sont reliés aux exigences que le système doit satisfaire. Un exemple d'un processus métier lié au système *AFS* sera *un utilisateur qui se connecte au système via une console.*

3. *Les règles métier:* Les règles sont dans la majorité des cas source de contraintes sur le système. La plupart portent sur l'architecture du système. De ce fait, il est important de comprendre ces contraintes. Un exemple de règle métier sur le système *AFS* est que le *System Manager ne doit permettre la connexion d'une console que si le processus d'authentification est bien déroulé.*

Dans le langage DSpec, les processus et les règles métier ne sont pas représentés par le langage. Ce choix est motivé par l'utilisation envisagée de la spécification du domaine capturée par les modèles DSpec. En effet, nous nous limitant à lister les ressources métiers afin de pouvoir les référencer dans les modèles de description des exigences et des contextes. Toutefois, ces processus et les règles métier peuvent être la source d'entités métier exprimé par le langage.

4.1.1.2 les entités du système

Lors de la conception d'un nouveau système logiciel, il est rare de démarrer à partir d'une page vide. En effet, il est souvent le cas de construire un nouveau système dont le domaine est le résultat d'autres systèmes existants que ce soit sur la partie matérielle ou logicielle. Pour ces systèmes, les entités du domaine représentent des composants logiciels et matériels ainsi que les autres éléments avec lesquels ils interagissent. Ainsi, les principaux aspects qu'il faut prendre en compte pour ces domaines sont: le système, les sous-systèmes, les modules, les connecteurs, les processus et les éléments matériels. Ces éléments seront représentés par le modèle du domaine créer au niveau des exigences à l'aide du langage DSpec. L'entité "système" est utilisée lors de la description des exigences, des cas d'utilisation et des scénarios. Les autres entités du système peuvent être référencées dans les exigences décrivant des contraints techniques sur le système étudié.

4.1.2 Représentation du vocabulaire du domaine

Dans cette section, nous introduisons une structure pour la représentation du vocabulaire du domaine capturé par le langage DSpec.

Le principal rôle de l'activité de spécification des exigences pour un système logiciel est de refléter les besoins du client. Cette spécification est la base sur laquelle les développeurs construisent le système qui répond au mieux aux attentes du client. Malheureusement, un problème récurrent avec les spécifications des exigences est qu'elles sont souvent imprécises et parfois incohérentes. D'ailleurs, une grande partie du travail de l'ingénieur IE (chargé d'étudier et d'analyser les exigences) est notamment de détecter ces incohérences [Kof 2004]. L'une des principales sources ce cette imprécision est la difficulté de comprendre et d'interpréter les intentions de leur auteur, ce qui cause des ambiguïtés dans les spécification produites.

Lors de la spécification des exigences, il est souvent le cas de que les les descriptions liées au comportement du système, à la qualité et à son apparence avec les descriptions des notions issues du domaine d'application soient mélangés [Edwards *et al.* 1995]. De ce fait, la signification de ces notions sont enfuies dans différents endroits tels que les documents de spécification et les scénarios. Pire encore, certaines notions peuvent

avoir plusieurs définitions, parfois conflictuelles, et différents synonymes sont utilisés pour décrire les mêmes notions. De ce fait, ces exigences sont difficiles à traduire correctement en des propriétés formalisées pour conduire un processus de vérification formelle.

Pour contrer ce problème, ou simplement limiter sont impact, nous avons besoin d'un langage facilitant l'activité de spécification d'exigences de meilleur qualité. Un tel langage doit pouvoir exprimer les besoins de la façon simple et précise à l'aide des phrases respectant une grammaire bien définie. Ainsi, pour exprimer une interaction entre le *System Manager* et l'*Equipment Manager* lors de la phase d'initialisation, on peut par exemple écrire:

- *SM asks the EquipentMgr for reachable consoles*
- *EquipementMgr sends the liste of registred consoles*
- *SM register is ready to receive loging request from reachable consoles*
- *Registred consoles asks the SM for login.*

Avec ces phrases simples, les interactions entre le système et les acteurs de l'environnement sont clairement définis et sont bien identifiées. Toutefois, pour garder cette clarté lors de la spécification, il est important de ne pas intégrer la définition des notions utilisées directement dans les exigences. Par exemple, les définitions des notions "*registred consoles*" ou "*loging requests*" ne sont pas définies ni la relation entre elles. Ainsi, nous avons besoin de définir un vocabulaire du domaine permettant de stocker les notions du domaine du système étudier avec leurs définitions et les relations qui peuvent les lier. De ce fait, nous proposons dans notre approche d'utiliser un langage de spécification des exigences qui s'appuie sur une spécification du domaine permettant de garder cette dichotomie entre les exigences et les définition des notions utilisées par les exigences. Pour l'exemple précédent, le vocabulaire du domaine devrait contenir des définitions telles que:

Reachable console Is a *console* that is *registred* with the *EquipmentMgr*

Logging request an *event* sent form the *console* to the *SM* in order to be able to execute missions. The login request contains the *id* of the sender *console* as parameter.

Les mots en italique dans les définitions représentent d'autres notions qui font partie du vocabulaire du domaine. Chaque notion définie dans le vocabulaire du domaine peut prendre différentes formes (singulier ou pluriel) et des synonymes. En plus des noms, le vocabulaire du domaine peut contenir des verbes. Toutefois, les verbes n'ont pas leurs définitions propres dans le vocabulaire du domaine, leurs sens dépendent du contexte du nom avec lequel ils sont utilisés. A titre d'exemple, "SM asks the EquipmentMgr" n'a pas le même sens que "Registred consoles asks the SM" même si les deux utiliser le verbe "ask".

Le vocabulaire du domaine est constitué à l'issue d'interviews entre les futurs utilisateurs du système et les experts du domaine métier. Lors de la rédaction des exigences

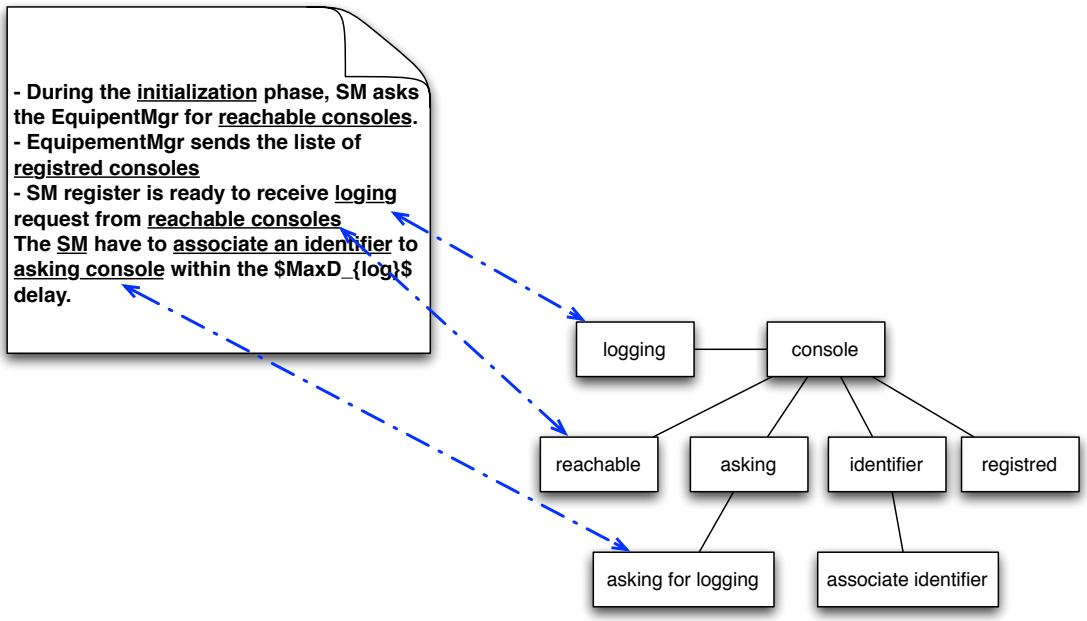

Figure 4.1 : Exigences avec des liens vers le vocabulaire du domaine

dans le langage approprié, l'auteur doit avoir accès à ce vocabulaire pour insérer les notions référencées directement dans ses phrases. Aussi, il doit pouvoir étendre le vocabulaire à tout moment en ajoutant de nouvelles notions (entités supplémentaires, verbes, actions) ainsi que leurs définitions en cas de besoin. Dans la figure 4.1, les phrases des exigences ont des liens vers des éléments du vocabulaire du domaine.

Dans la section suivante, nous décrivons la structure du vocabulaire du domaine proposé pour stocker les notions du domaine, leurs définitions ainsi que les relations qui les lient.

4.2 Modèle conceptuel pour la spécification du domaine

Le métamodèle du langage DSpec est organisée en cinq packages:

Terminology: ce package contient les termes ainsi que leurs relations avec un thesaurus. Le thesaurus est un dictionnaire conceptuel structuré. Il contient les informations morphologiques sur les termes utilisés, telles que les différents formes qu'il peut prendre (singulier/pluriel, conjugaisons ...) ainsi que les informations sémantiques. Dans notre langage, nous nous appuyons sur un thesaurus existant appelé WordNet. (voir la section 4.2.2)

Phrases: ce package introduit au langage les constructions nécessaires pour la rédaction des phrases dans un langage structuré. Les termes utilisés dans ces phrases sont des références vers les termes proposés par le thesaurus.

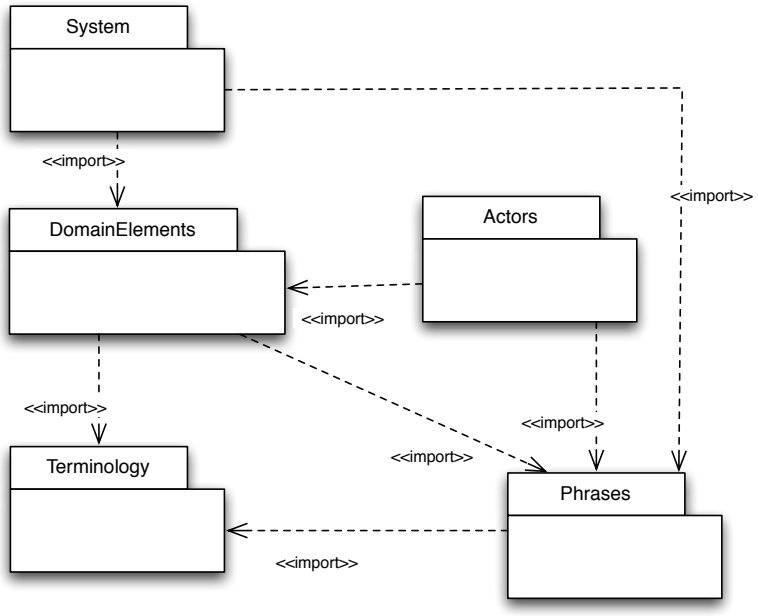

Figure 4.2 : Organisation du métamodèle de spécification du domaine

DomainElements ce package contient l'ensemble des notions relatives au domaine étudié. Pour chaque notion, on peut attacher des phrases pour exprimer les différentes constructions possible pour chaque notion. Les éléments du domaine peuvent avoir des liens entre eux.

Actors Permet la définition des acteurs.

System ce package permet de définir le système en développement ainsi que les principaux composants. Cette définition permet de référencer le système dans les exigences et les scénarios du contexte.

Ainsi, d'une façon générale, les éléments du vocabulaire définis au sein du langage DSpec sont des *DomainElements* qui ont des noms et des *phrases* associées comme description.

Dans la suite de cette section, nous allons détailler le contenu de chaque package et expliquer le rôle qu'il joue pour la production d'une spécification du domaine exploitable dans le cadre de la formalisation des exigences et des contextes du système étudié.

4.2.1 DomainElements package

Le package principal définissant l'ensemble des éléments du domaine est le package *DomaineElements*. Chaque élément du domaine fait partie du vocabulaire du domaine et contient une ou plusieurs *statements* (phrases qui référence cet élément du domaine). La description des métaclasses de ce package (présenté à la figure 4.3) est donnée comme suit:

DomainStatement représente une description d'un élément du domaine du système.

DomainVocabulary est une structure qui regroupe les éléments constituant un conteneur pour ces derniers. Les éléments contenus dans le vocabulaire du domaine sont soit des éléments du domaine (*DomainElements*) soit des acteurs (*Actors*).

DomainElement est une structure regroupant un ensemble de phrases *DomaineStatements* dans lesquelles l'élément du domaine en question apparait. En d'autre terme, les phrases dont le "sujet" à le même *Terms::Noun* que l'attribut *name* du *DomainElement* considéré.

DomainElementAssociation Définit les relations entre les différents éléments du domaine.

NounLink Représente un lien qui pointe vers le nom (*Terms::Noun*) utiliser comme nom de l'élément du domaine.

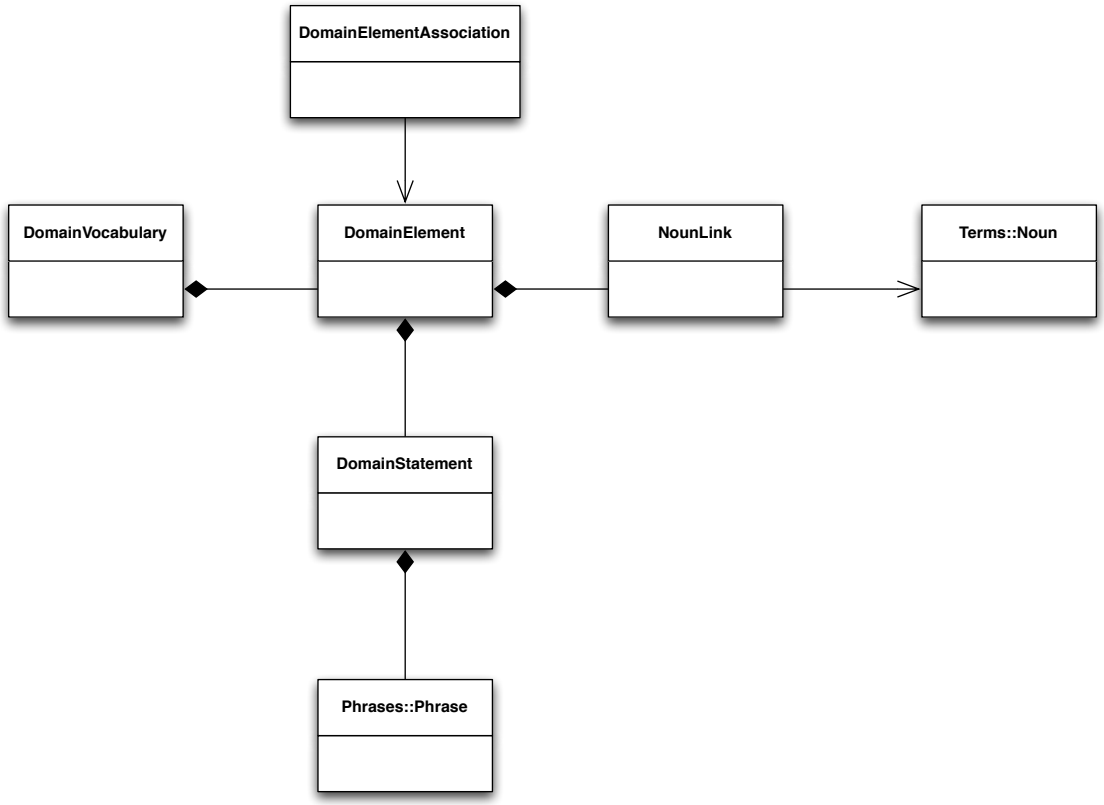

Figure 4.3 : Métamodèle des éléments du domaine et liens entre les packages

Pour illustrer les métaclasses décrites ci-haut, nous considérons le cas d'étude *AFS*. La figure 4.4 schématise les différents éléments qui entre dans la définition des éléments du domaine du cas d'étude *AFS*.

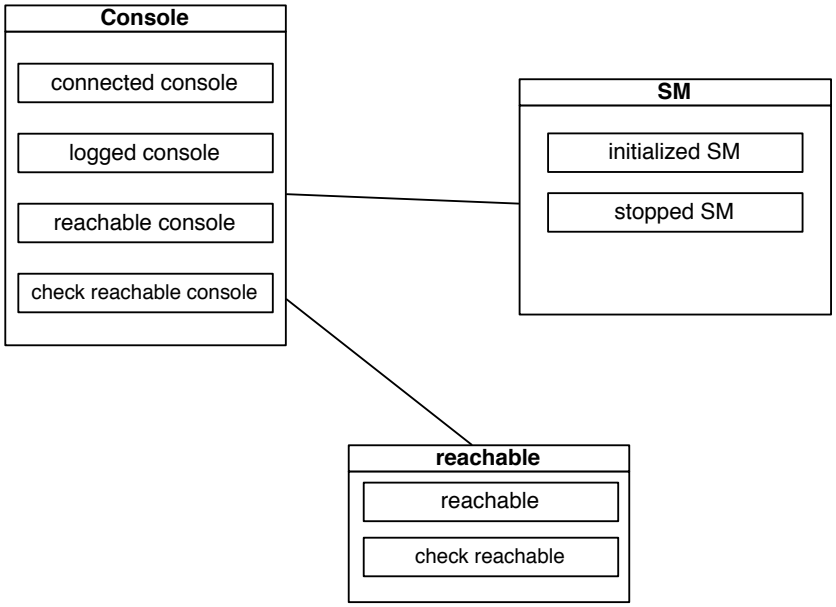

Figure 4.4 : Vue partielle des éléments du domaine de l'*AFS*

4.2.2 Terminology Package

Ce package contient la définition des termes utilisés pour la spécification des exigences. La figure 4.6 illustre le contenue du package *Terminologie*. Chaque terme est présenté avec sa définition ainsi que les relations sémantiques qu'il partage avec d'autres termes. Les relations sémantiques utilisés sont :

- *TermSpecialisationRelation*: cette relation décrit les termes sous la forme d'une structuration hiérarchique. La figure 4.5 illustre cette relation pour les noms ainsi que pour les verbes.
- *HasSynonym*: est une relation de type synonyme.
- *HasHomonym*: est une relation de type homonyme.

En effet, les relations entre les termes précisent leurs sémantiques. Cette structuration des termes employés dans les exigences est représentée par une ontologie. Nous nous basons sur une ontologie existante, appelée WORDNET pour les définitions et les relations entre les termes usuels du langage. L'utilisateur précise les termes nouveaux liés au domaine métier dans la spécification du domaine et s'appuie sur WORDNET pour les autres termes usuels du langage.

WORDNET[1] est le résultat de la combinaison d'un dictionnaire de définitions et un thesaurus. Cette base de donnée lexicale fait le lien entre les noms, verbes, adjectifs et adverbes de la langue anglaise [Miller 1995, Fellbaum 1998, Wolter *et al.* 2008]. Les

[1] http://wordnet.princeton.edu/

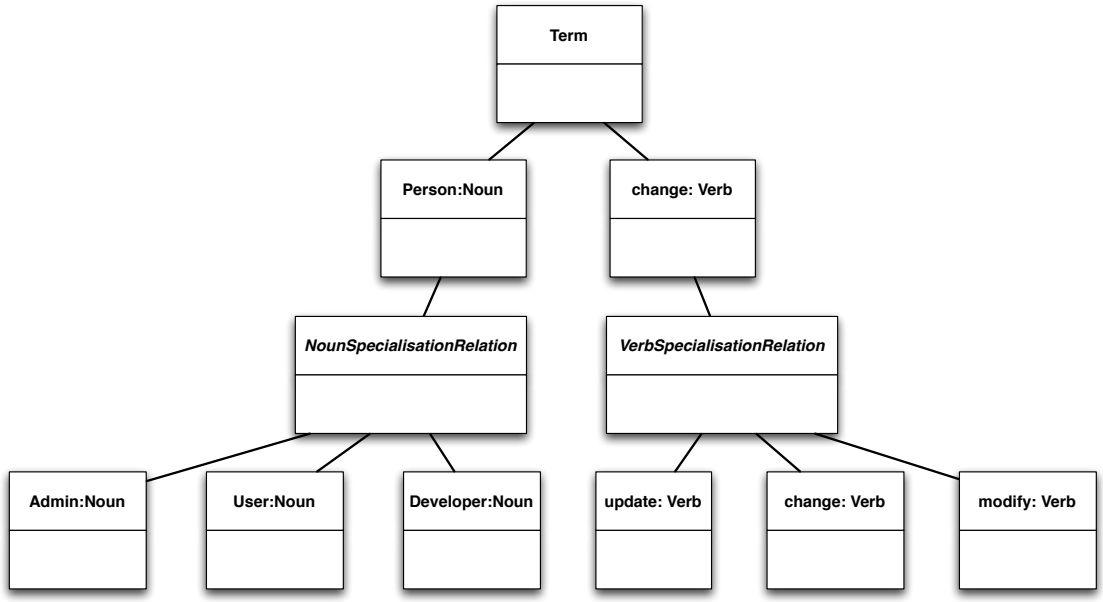

Figure 4.5 : Exemple de relations de spécialisation entre les termes

termes sont groupés dans des sous-ensembles de synonymes (*Synset*) dans lequel sont groupés les termes exprimant un même concept. Ces sous-ensembles sont reliés entre eux par des relations sémantiques et lexicales. Le principal type des relations existantes dans WORDNET est la relation synonyme. Chaque *synset* contient une définition succincte ainsi qu'une ou plusieurs phrases illustrant l'utilisation du terme en question.

D'autres bases de données de ce type existent pour d'autres langages européens (par exemple, *Wortschatzlexikon* [2] pour l'Allemand). Le projet EuroWordNet[3] propose une base de données Wordnet multilingue pour plusieurs langues (Allemand, Italien, Espagnole, Germanique, Français, Tchèque et Estonien).

Dans le cadre du langage URM, WORDNET est utilisée pour stocker les termes utilisés dans les exigences avec les relations sémantiques qu'ils partagent avec les autres termes de la base de données. L'utilisateur est appelé à enrichir cette *terminologie* par les termes métiers, leurs définitions ainsi que leurs relations avec les autres termes.

L'intérêt d'utiliser une telle ontologie généraliste réside dans le fait de pouvoir référencer l'ensemble des termes utilisés dans la description des exigences et des scénarios des contextes afin de réduire les risques d'ambiguïté sur leurs sens. En effet, chaque élément du domaine est décrit avec sa description au sein de fichiers DSpec. Pour chaque élément on identifie l'ensemble des phrases dans lequel il est le sujet (*Terminology::Noun*). Chaque phrase est basée sur les termes existant dans le thesaurus généraliste (WORDNET). Ce thesaurus stocke les termes avec leurs changements (inflexions), relations avec les autres termes (homonymes, synonymes) et les classes selon leurs natures grammaticales.

[2] http://Wortschatz.uni-leipzig.de
[3] Projet: LE-2 4003 & LE-4 8328; http://www.illc.uva.nl/EuroWordNet/index.html

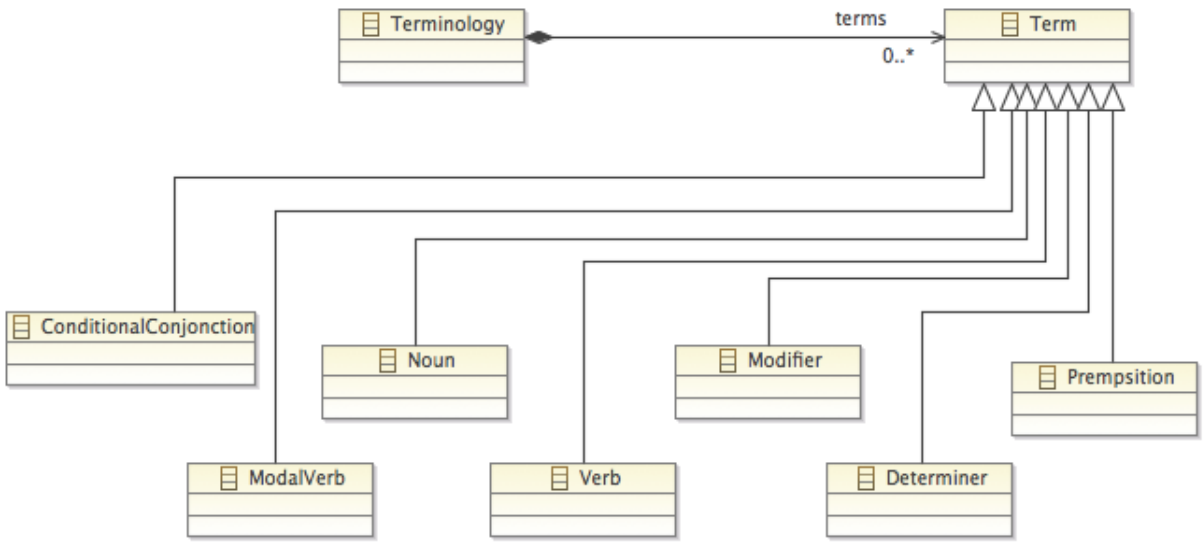

Figure 4.6 : Présentation du package *Terminology*

Ce mécanisme permet d'utiliser non seulement les phrases identiques, mais des phrases similaires lors de la réutilisation des exigences.

Nous illustrons l'utilisation d'une telle spécification du domaine sur le cas d'étude présentés à la section 3.1 dans le chapitre 9.

4.3 Discussion et synthèse

Au cours du processus de développement de systèmes logiciels, plusieurs types de modèles sont produits pour capturer les différents aspects importants du futur système. À titre d'exemple, une approche orientée objet utilisant le langage UML génère plusieurs diagrammes UML. La nature hétérogène de ces modèles nous pousse à nous poser la question de la cohérence entre ces différents artefacts. Après l'étude de l'état de l'art, nous avons pu identifier une solution consistant à construire une spécification du domaine. Celle-ci permet de capturer les différentes notions et entités du domaine étudié afin de pouvoir les référencés plus tard dans les modèles de contextes et les exigences.

Dans ce chapitre, nous avons présenté un langage permettant de construire la spécification du domaine étudié. Cette spécification du domaine nous permettra de définir les éléments constituant la base de connaissance liée au système étudié. Nous avons organisé cette spécification en trois packages: un pour les acteurs, un autre pour les composants du système étudié et un dernier pour regrouper l'ensemble des notions du domaine (messages, arguments, attributs...).

Pour les autres termes utilisés dans la spécification des exigences, nous avons réutilisé une ontologie généraliste existante afin de pouvoir déterminer le sens des termes employés et lever les ambiguïtés. De cette façon, nous disposons d'une base commune, référençant

les entités et les termes utilisés lors de la spécification des exigences et des scénarios des contextes. Dans le chapitre suivant, nous proposons un langage permettant la spécification des scénarios d'interaction entre le modèle du composant étudié et son contexte tout en s'appuyant sur les entités référencées par une spécification du domaine DSpec.

5 Capture et Formalisation des Contextes

5.1 Présentation du langage XUC

Nous avons vu dans le chapitre 3 que le langage CDL a été proposé dans l'optique de capturer et de formaliser les contextes et les propriétés à vérifier. Bien que les résultats obtenus ont été prometteurs [Dhaussy *et al.* 2009], l'intégration du langage CDL dans un processus de développement industriel n'a pas atteint le niveau escompté. Dans la section 1.2, nous avons identifié les principales raisons qui limitent l'adoption du langage CDL par les ingénieurs concernés par la vérification. Dans ce chapitre nous proposons un langage, dit orienté utilisateurs, dont le but est de faire le lien entre les modèles manipulés par les ingénieurs dans les premières étapes du processus de développement et les modèles CDL.

En effet, nous proposons que le langage CDL soit considéré comme langage interne, c'est à dire généré automatiquement à partir de modèles de plus hauts niveau, manipulés au cours du processus de développement. Notamment, les cas d'utilisation et les scénarios, qui sont largement répandus et bénéficient d'une popularité dans le milieu industriel [Uchitel *et al.* 2004, Dalal A & Uchitel 2007].

5.1.1 Motivation de la proposition des XUC

La motivation derrière la proposition d'un nouveau DSL pour la capture et la formalisation des contextes résulte de l'écart sémantique entre les modèles manipulés par les concepteurs au cours des activités de spécification et ceux nécessaires aux activités d'analyses formelles. Pour illustrer, nous allons donner un exemple de ces deux types de modèles tels qu'ils sont manipulés aujourd'hui.

En guise d'illustration, considérons le cas d'étude AFS présenté dans la section 3.1 à la page 35.

Les différents cas d'utilisation sont décrits dans les documents de spécification soit textuellement soit à l'aide de diagrammes de séquence UML2. Dans ces cas d'utilisation, plusieurs acteurs interagissent avec le composant que l'on veut vérifier (le *SystemManager*) selon différents scénarios d'interactions. Pour illustrer, nous présentons ici les deux types de descriptions rencontrées dans les documents de spécification de l'AFS. La figure 5.1 présente la description textuelle du cas d'utilisation *Initialization*.

À partir de cette description, un modèle CDL est manuellement produit pour formaliser les interactions entre le *SM*, l'*EquipmentMgr* et les consoles (*HMI*). En effet, les

Use Case: Initialization
Actors: Equipment Manager
Intention: During the initialization, the *SM* asks the *Equipement Manager* for reachable consoles to be ready to process login requests from.
Main Success Scenario:

s1. The *SM* sends a *goInitialization* message to the *Equipment Manager*

s2. The *Equipement Manager* sends a goInitializationAck message to the SM and an initConsole message for each reachable HMI

s3. *HMI* sends an *initConsoleAck* to *Equipment Manager*

s4. *Equipment Manager* sends a *addConsole* message to *SM* with the console id

s5. Foreach *addConsole* message received from the *Equipment Manager*, the *SM* sends an ack message to the *Equipment Manager*.

Use case ends in success.

Exceptions:

e1: Console initialization failed

related step: s3
description: The console can not be initialized
related handler: none
outcome: «failure»

Figure 5.1 : Description textuelle du cas d'étude *Initialization*

scénarios sont exprimés en programme CDL conformément à la syntaxe du langage. Les enchaînements d'envoi et de réception de messages sont regroupés dans des constructions CDL faisant référence à des *activités*. Une *activité* regroupe plusieurs événements et les séquencements entre les interactions sont exprimés avec différents opérateurs (voir la section 3.3 pour la description des opérateurs de composition CDL). La figure 5.2 montre un extrait du programme CDL correspondant au cas d'utilisation *Initialization*.

Code CDL

```
CDL Initialisation
{
  events{
    //Déclaration des événements: Voir l'annexe~A.2
  }
  par MAIN{
    HMI1,
    HMI2,
    EquipMgr
  }

  //Déclaration des interactions
  interaction HMI1_init_op1          { HMI1_s_initConsolAck_OK }
  interaction HMI2_init_op1          { HMI2_s_initConsolAck_OK }
  interaction HMI1_init_op2          { HMI1_s_initConsolAck_KO }
  interaction HMI2_init_op2          { HMI2_s_initConsolAck_KO }
  interaction EquiMgr_Init_op1 {EquiMgr_s_goInitAck_OK, EquiMgr_s_initConsol,
                                EquiMgr_s_addConsole}
  interaction EquiMgr_Init_op2 {EquiMgr_s_goInitAck_KO}
  interaction EquiMgr_Init_op3 {EquiMgr_r_addConsolAck_KO}
  interaction EquiMgr_Init_op4 {EquiMgr_r_addConsolAck_OK}

  alt HMI1
    {
          HMI1_init_op1,
          HMI1_init_op2,
    }
  alt HMI2
    {
          HMI2_init_op1,
          HMI2_init_op2,
    }
  alt EquiMgr
    {
      EquiMgr_Init_op1,
      EquiMgr_Init_op2,
      EquiMgr_Init_op3,
      EquiMgr_Init_op4
    }
} // fin CDL
```

Figure 5.2 : Extrait du programme CDL *Initialization*

Au sein du programme CDL, le comportement de chaque acteur est considéré comme étant un enchaînement de scénarios qui décrivent les interactions entre le modèle à valider et les acteurs de l'environnement. Les comportements de chaque acteur sont composés en parallèle pour générer l'ensemble des enchaînements d'événements possibles. Cela implique qu'à partir d'une description de haut niveau des cas d'utilisations du système, l'utilisateur est appelé à identifier le comportement de chaque acteur de l'environnement

afin de le formaliser sous la forme d'un scénario CDL. Ce processus n'est pas évident, surtout lorsque le système est fortement couplé avec son environnement où lorsque le nombre d'acteurs est très grand.

D'un autre côté, produire une description exhaustive des cas d'utilisation en terme de modèle CDL s'avère être une tâche compliquée du fait que le langage CDL s'appuie sur des scénarios simples. En effet, une spécification en terme de scénario est une spécification partielle. Chaque scénario ne représente qu'un ensemble partiel d'interactions. De ce fait, pour pouvoir couvrir l'ensemble du comportement d'un acteur interagissant avec le système étudié, il faut modéliser son espace d'état exhaustif sous la forme d'une machine à états.

D'un point de vue industriel, le langage CDL est perçu comme étant un langage de bas niveau, contraignant et difficile à appréhender sur des modèles complexes. Il est vu en tant que langage de bas niveau du fait de sa notation qui demande à lister l'ensemble des événements échangés entre le système et son environnement (voir la section A.2). Du point de vue des experts métier qui sont supposés produire des modèles CDL pour la spécification des contextes, CDL est considéré difficile à appréhender du fait de l'effort important nécessaire pour identifier et spécifier le comportement des acteurs de l'environnement. En effet, produire une spécification du contexte à l'aide du langage CDL exige une description détaillée du système à valider dès les premières étapes du processus de développement [Dhaussy *et al.* 2009].

Nous cherchons donc à avoir une description de haut niveau, plus proche de la compréhension des concepteurs et des modèles qu'ils ont l'habitude de manipuler. Dans le reste de ce chapitre, nous allons présenter le langage XUC proposé pour surmonter les difficultés qui limitent l'intégration du langage CDL aux processus de développement industriels.

5.1.2 Description informelle des XUC

Nous avons proposé des cas d'utilisation étendus (eXtended Use Cases) dans le but de réduire l'écart entre les modèles utilisés en contexte industriel et les modèles CDL.

Les objectifs posés lors de la proposition des XUC sont les suivantes :

1. Simplicité de la syntaxe pour rester proche de la compréhension des concepteurs et des modèles qu'ils ont l'habitude de manipuler (cas d'utilisation UML + langage naturel contrôlé).

2. Être suffisamment expressif afin de permettre aux ingénieurs de spécifier en détail leurs systèmes et ne pas contraindre la conception (permettre les différents type d'interactions et la composition de ces interactions).

3. Disposer d'une sémantique claire et précise permettant d'analyser automatiquement (à l'aide d'outils) les modèles conçus.

Un modèle XUC a pour but de capturer et formaliser les interactions entre le système à vérifier et son environnement dans les différents cas de son utilisation. De ce fait, le

langage XUC est proposé sous la forme d'une extension des Use Cases UML2 [OMG 2007] pour décrire plus précisément les interactions qui se déroulent au sein de chaque cas d'utilisation.

Ces interactions s'effectuent sous la forme de scénarios. Au sein d'un cas d'utilisation, un scénario est présenté pour décrire l'enchaînement attendu des événements réalisant le cas d'utilisation. Ce scénario, est appelé dans la littérature un *scénario nominal* [Cockburn 2000, Jacobson 2004, Övergaard & Palmkvist 2004], et il est éventuellement complété par des scénarios alternatifs. Ces derniers précisent la façon dont le système réagit dans l'éventualité où une exception est produite durant le déroulement du scénario nominal.

Ainsi, en plus des scénarios nominaux, un modèle XUC permet de décrire les exceptions qui peuvent surgir à n'importe quelle étape du scénario nominal ainsi que les étapes nécessaires permettant au système de gérer ces exceptions (appelés *handlers*).

L'idée d'ajouter les scénarios exceptionnels dans le corps des cas d'utilisation n'est pas idée nouvelle. Plusieurs travaux ont montré l'importance de considérer les exceptions qui peuvent le déroulement des interactions normales du système avec son environnement [Almendros-Jimenez & Iribarn 2005, Shui *et al.* 2005, Mustafiz *et al.* 2008]. Plus particulièrement, le besoin de considérer les scénarios exceptionnels apparait dans le cas des systèmes réactifs. Pour ces systèmes, une exception non identifiée peut, potentiellement, conduire à des spécifications incomplètes au cours des étapes d'analyse du modèle du système ce qui peut résulter vers une implémentation dont les fonctionnalités ne sont pas complètes [Shui *et al.* 2006].

Dans l'approche proposée dans ce mémoire, nous nous sommes inspirés de ces travaux pour étendre les cas d'utilisation UML avec des scénarios alternatifs. Le corps de chaque scénario est exprimé sous la forme d'une suite d'étapes (*Step*). Chaque étape est décrite sous forme textuelle en respectant une grammaire précise. Le but recherché par cette structuration du langage pour exprimer les étapes d'un XUC est de pouvoir extraire des modèles avec une sémantique bien définie directement des phrases saisies par l'utilisateur. Ces modèles vont permettre, par la suite, l'application d'une transformation produisant des modèles CDL. Ce processus est détaillé dans le chapitre 7.

5.1.3 Structure d'un XUC

Chaque fichier *.xuc* contient la description d'un cas d'utilisation du système étudié. Ce cas d'utilisation est construit à l'aide des différents scénarios proposés par le langage XUC. L'objectif de chaque cas d'utilisation étendu est de capturer les interactions qui peuvent avoir lieu les acteurs participants à ce cas d'utilisation et le composant sous validation. Ces interactions sont exprimées par les différents scénarios contenus dans un XUC. La figure 5.3 présente la structure d'un XUC et montre les différents scénarios que peut contenir ce dernier pour décrire les interactions entre le système et son contexte.

Selon la sémantique des XUC (présentée dans la section 5.2, les différents scénarios capturés par un XUC sont reliés afin d'exprimer les interactions détaillées entre le système

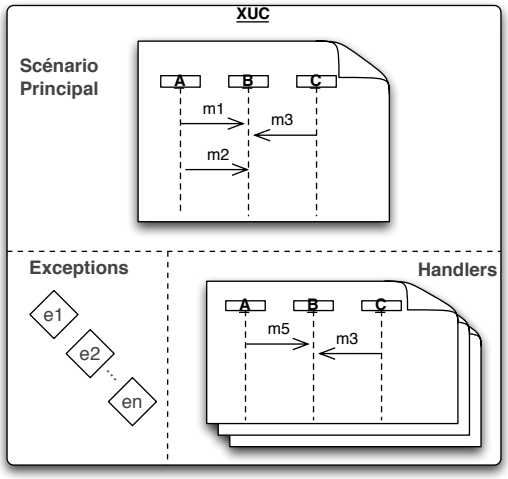

Figure 5.3 : Structuration des scénarios au sein d'un XUC

et son environnement. Ainsi, on peut construire un diagramme d'activité reflétant le comportement complet de chaque XUC. Dans ce diagramme d'activité, les noeuds de contrôle ont la même sémantique que les noeuds de contrôle du modèle XUC. Les noeuds (*UML::Activity*) du diagramme d'activité représentent les différentes étapes (*XUC::Step*) du cas d'utilisation. Pour interpréter ces étapes, nous nous appuyons sur la spécification du domaine capturée par les fichiers *.dspec*. La description détaillée de cette transformation est présentée dans la section 7.1.

5.2 Syntaxe et sémantique

5.2.1 Syntaxe abstraite

Le diagramme de la figure 5.4 présente la syntaxe abstraite du langage XUC. La classe *eXtendedUseCase* est sous classe de la classe *uml::UseCase*. Elle contient un scénario (*mainScenario*) décrivant les interactions entre les acteurs de l'environnement et les composants du système. L'attribut *intention* permet de décrire le cas d'utilisation de façon informelle sous forme textuelle. Le scénario d'un use case (*UCScenario*) détaille les échanges Système/Environnement sous la forme d'une succession d'étapes (*Step*) reliées entre elles par des transitions (*Transition*) et des noeuds de contrôle (*ControlNode*). Différents types de noeuds sont proposés (*ControlNodeKind*) pour composer les étapes sous forme de scénarios.

En plus de la spécification du scénario principale, un ou plusieurs scénarios exceptionnels, appelés *Handlers*, peuvent être spécifiés au sein d'un XUC. Un *handler* est défini sous forme d'un scénario d'étapes et de transitions et peut être lié à une ou plusieurs exceptions (réutilisation des scénarios).

Pour chaque scénario possible, on définit un *Outcome* pour préciser le résultat du cas

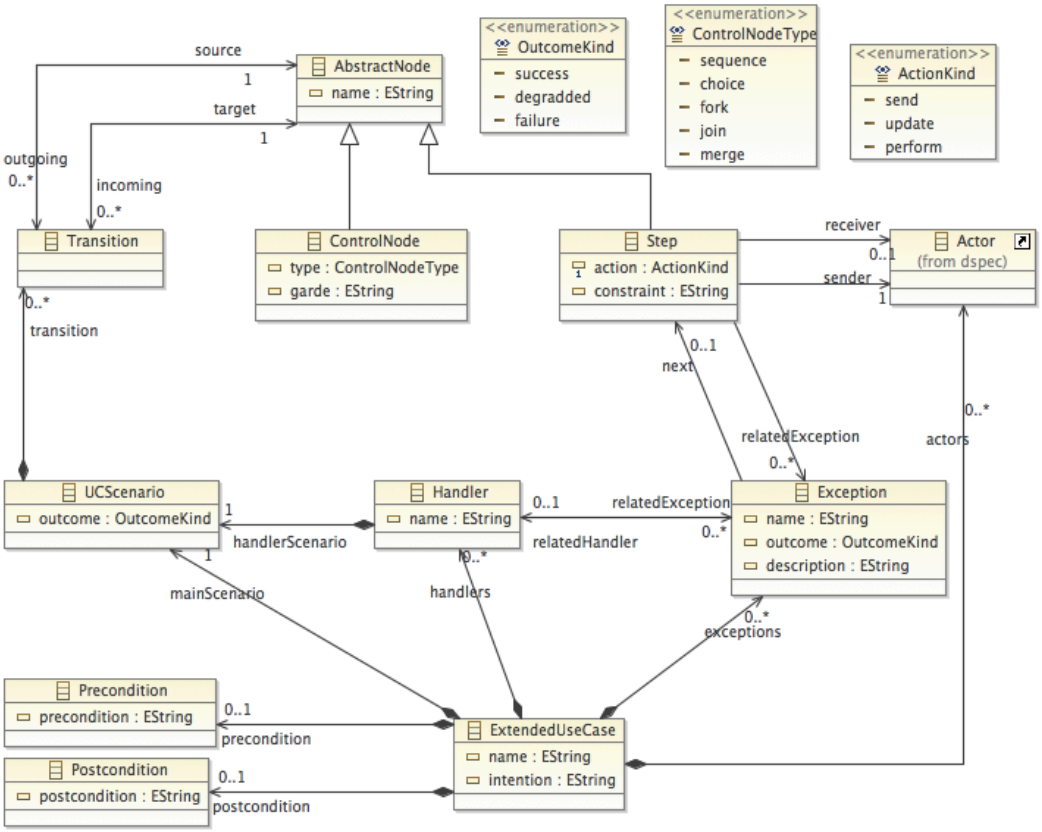

Figure 5.4 : Métamodèle des XUC

d'utilisation (si celui-ci s'est bien déroulé ou pas). En effet, un scénario peut terminer en succès parce que le use case s'est déroulé comme prévu ou en échec, car une exception inattendue a fait en sorte que le cas d'utilisation n'atteint pas son objectif prévu. Un scénario peut aussi terminer en mode dégradé dans certains cas. La section 5.2.2 précise la sémantique des scénarios d'un XUC tandis que la section 5.2.3 décrit la manière avec laquelle les étapes d'un scénario proposent de formaliser les échanges qui peuvent avoir lieu au cours de chaque étape.

5.2.2 Sémantique des scénarios

Dans la section précédente, nous avons présenté la syntaxe abstraite d'un cas d'utilisation étendu (XUC) et les concepts qui le composent. Nous avons mentionné que plusieurs scénarios peuvent être définis au sein d'un même XUC pour exprimer les interactions entre les systèmes et son contexte. Dans cette section, nous allons définir la sémantique de ces scénarios et la manière avec laquelle nous les interprétons et surtout, les implications de cette interprétation sur l'utilité des XUC au sein d'un processus de vérification formelle.

En regard de la littérature, trois types de définitions de la sémantique des langages à base de scénarios sont proposés: les sémantiques informelles, algorithmiques et ab-

straites [Uchitel 2003].

La première catégorie correspond aux langages ne disposant pas d'une sémantique précise utilisés dans le contexte des méthodes de développement informelles telles que les méthodes de développement basées sur le langage UML (Unified Software Development Process [Booch *et al.* 1999]). La deuxième catégorie regroupe les approches dont la sémantique des scénarios est implicite, c'est-à-dire qu'elle est donnée par un algorithme de transformation. En effet, en utilisant un algorithme de traduction d'un scénario vers une autre notation peut fournir une interprétation précise si la notation cible dispose d'une sémantique bien définie [Carroll 1995]. Toutefois, ce type d'interprétation est opérationnelle, car elle ne permet pas de donner une signification intuitive et abstraite du scénario [Uchitel 2003]. Dans ce cadre, on trouve les approches qui proposent de traduire les scénarios vers les machines à états [Koskimies & MLkinen 1994, Harel & Kugler 1999, Whittle & Jayaraman 2006, Ziadi *et al.* 2009]. La troisième catégorie de sémantique consiste à définir une sémantique formelle abstraite pour les scénarios. La différence avec la sémantique donnée à l'aide d'un algorithme de traduction réside dans le fait qu'elle est abstraite, c'est-à-dire définie en utilisant des langages formels tels que l'algèbre des processus [Reniers 1999, Genest & Muscholl 2005], les ordres partiels [Alur *et al.* 2000], les automates de büchi [Ladkin & Leue 1995] ou les réseaux de pétri [Heymer 2000].

Au sein d'un XUC, l'ensemble des événements échangés entre le système et les acteurs intervenants dans ce XUC se déroule selon l'ordre défini par le scénario des étapes. Pour spécifier cet enchainement, nous nous appuyons sur cette dernière catégorie de sémantique, c'est-à-dire définir une sémantique formelle abstraite dédiée. Aussi, afin de pouvoir donner un sens précis et rigoureux aux comportements capturés par les Xuc, nous avons défini une sémantique opérationnelle pour traduire les spécifications XUC vers des programmes CDL.

D'un point de vu structurel, le corps d'un XUC peut être vu comme un graphe orienté avec deux types de noeuds: Les étapes (*step*) et les noeuds de contrôle, connectés par des transitions. La figure 5.5 montre une représentation graphiques du scénario capturé par le XUC de la figure 5.1.

D'un point de vue dynamique, le comportement d'un XUC est décrit par un ordre spécifique d'exécution des actions contenues dans les étapes référencées. À l'instar des diagrammes d'activités, cet ordre dépend de la propagation d'un *jeton* qui, initialement, est détenu par le noeud de départ S_0. Quand un noeud *step* reçoit ce jeton, il devient actif et il commence son exécution. À la fin de cette exécution, le jeton est passé à sa transition sortante. Au cours de l'exécution d'un XUC, sa structure reste inchangée et seule la position du jeton de contrôle change. De ce fait, la sémantique du comportement d'un XUC peut être décrite par un ensemble de règles de progression qui dictent le mouvement du jeton à travers le corps du XUC.

Pour des raisons de simplification de l'écriture, chaque noeud du corps du XUC sera identifié par un label unique noté l. Un noeud identifié par un label l est alors écrit $l : N$, où N peut être n'importe quel noeud dans S sauf le noeud de départ S_0. Le label d'un noeud est utilisé pour faire référence au noeud correspondant s'il est rencontré plus tard

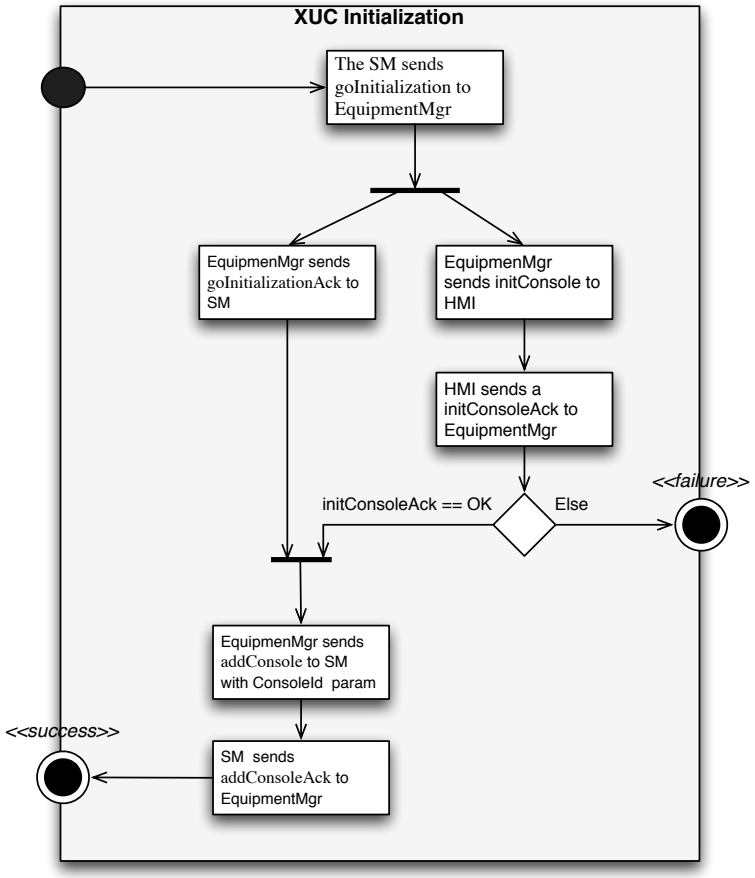

Figure 5.5 : Représentation graphique du scénario d'initialisation

lors de l'exécution du XUC.

Ainsi, un XUC est définit comme suit:

Définition 1 (eXtended Use Cases) : Un use case étendu est décrit par un tuplet
$(MS, E, H, Ac, St, Ps, T, \lambda)$ où:

- $MS = S \cup T$ représente le scénario principal du use case permettant d'atteindre l'objectif attendu par ce use case sous forme d'une succession d'étapes. Avec $S = St \cup Ps$ est l'ensemble de ces étapes. T représente l'ensemble des transitions reliant les étapes entre elles à travers les noeuds suivant la relation $\lambda = T \times S \cup Ps \times T$.
- E est l'ensemble des exceptions qui peuvent avoir lieu lors du l'exécution du scénario principale. Une exception est liée à une étape.
- H représente l'ensemble des étapes exécutées suite au déclenchement d'une exception.
- Ac est l'ensemble des acteurs intervenant dans le XUC.
- St est l'ensemble des étapes. Chaque étape $s \in S$ est un triplet (tr, m, re) d'un *message* $m \in M$, un émetteur tr et un récepteur re. M est l'ensemble de tous les messages. Soit Ev l'ensemble des événements et chaque événement est une paire contenant un *type* et un *message*:$(k, m) \in \{!, ?\} \times M$.
- $Ps = S_0 \cup EP \cup Fk \cup Jn \cup Ch$ est l'ensemble des noeuds de contrôle entre les étapes. Avec S_0 le noeud de départ, S_f, le noeud de la fin, Fk le fork, Jn pour le joint et Ch représente le choix.

Les différentes constructions possibles au sein du corps d'un XUC sont définies comme suit:

$$
\begin{aligned}
XUC ::= \quad & e \\
| \quad & S_0 \longrightarrow N \\
N ::= \quad & e \\
| \quad & l : S_f \\
| \quad & l : Merge(N) \\
| \quad & l : Fork(N_1, N_2) \\
| \quad & l : x.Join(N) \\
| \quad & l : Decision(\langle g \rangle N_1, \langle \neg g \rangle N_2)
\end{aligned}
$$

Le symbole e représente une activité vide alors que $\longrightarrow \in T$ représente une transition. Parmi les constructions possibles des noeuds d'un XUC, on trouve:

- $l : S_f$ représente le noeud final

- $l : Merge(N)$ (resp. $l : x.Join(N)$) est la définition du noeud *merge* (resp. *join*) où plusieurs transitions entrantes se rencontrent au niveau de ce noeud pour ensuite se diriger vers le noeud N. Le noeud *join* est un noeud de synchronisation, c'est-à-dire que le *jeton* ne peut passer vers le noeud N que si toutes les transitions sont marquées. Un noeud N est dit *marqué* à un moment donnée de l'exécution du XUC si et seulement s'il dispose du jeton. Il est alors noté $\bar{N}$. La variable x dénote le nombre des transitions incidentes au noeud *join*.
- $l : Fork(N_1, N_2)$ représente le noeud *fork* (où parallèle). Les paramètres N_1 et N_2 représentent les sous termes correspondants à la cible des transitions sortantes.
- $l : Decision(\langle g \rangle N_1, \langle \neg g \rangle N_2)$ représente le noeud décision où un choix est opéré entre deux noeuds N_1 et N_2 suite à l'évaluation d'une variable booléenne g.

Lors du déroulement du scénario décrit par un XUC, le *jeton* traverse les noeuds à partir du noeud de départ S_0 et s'arrête quand il atteint le noeud final S_f en suivant les transitions. Les transitions entre les noeuds d'un XUC ne portent aucune action. Le rôle des transitions est de faire transiter le jeton du noeud source vers le noeud cible. Une transition est dite marquée si sont noeud source est marqué par le jeton. Le jeton est représenté par une barre au dessus de l'identifiant du noeud qui le détient (exemple: $\bar{N}$). Les règles de progression du jeton à travers les noeuds d'un XUC sont décrites comme suit:

Initial: la règle `Init1` stipule que l'expression $i \longrightarrow N$ conduit à la transition $\bar{i} \longrightarrow N$ sans aucune action observable. Ainsi, activer un XUC revient à placer un jeton au niveau de son nœud initial S_0. De même, si le nœud initial est marqué, le marquage se propage vers le nœud suivant sans action observable (`Init2`).

$$\frac{}{\bar{i} \longrightarrow N} \text{ [init1]} \qquad \frac{\bar{i} \longrightarrow N}{i \longrightarrow \bar{N}} \text{ [init2]}$$

Final: la règle `final` montre que le marquage du jeton est supprimé après le passage par un noeud final. Autrement dit, l'exécution du XUC se termine et n'est plus actif dès qu'il atteins son nœud final.

$$N\ [\overline{l : S_f}] \longrightarrow N \quad \text{[final]}$$

Fork: si un nœud *fork* est marqué, la propagation du jeton passe aux nœuds suivant simultanément.

$$\frac{\overline{l : fork(N_1, N_2)}}{l : fork(\bar{N}_1, \bar{N}_2)} \text{ [fork]}$$

Join: tel que c'est précisé par le standard UML2 [OMG 2007]. La traversé d'un nœud *joint* ne se fait que lorsque tous les flux qui lui sont incidents soit marqués.

$$\frac{\overline{l : x.join(N)}^{x}}{l : x.join(\bar{N})} \quad x \geq 1 \text{ [join1]}$$

La règle `join2` montre que si le nœud *join* référence x flux incidents, la propagation du jeton vers le nœud N ne se produit que si tous les flux sont marqués. La règle join2, montre la possibilité de passer de $\overline{l : x.join(N)}$ à $l : x.join(\bar{N'})$, si $N \longrightarrow N'$.

$$\frac{N \longrightarrow N'}{\overline{l : x.join(N)}^{n} \longrightarrow l : x.join(\bar{N'})} \text{ [join2]}$$

Merge: à la différence du *join*, le nœud *merge* permet d'unifier plusieurs flux pour passer vers un nœud N sans exiger leurs synchronisation. la règle `merge1` montre la propagation d'un jeton vers le nœud N.

$$\frac{\overline{l : merge(N)}^{n}}{l : merge(\bar{N})^{n-1}} \text{ [merge1]}$$

La règle `merge2` montre l'évolution du marquage vers l'expression $\overline{l : merge(N')}^{n}$ s'il y a une transition possible telle que $N \longrightarrow N'$.

$$\frac{N \longrightarrow N'}{\overline{l : merge(N)}^{n} \longrightarrow \overline{l : merge(N')}^{n}} \text{ [merge2]}$$

Decision: les règles `decision1/2` montrent la propagation du jeton à travers un nœud de décision. En effet, le marquage se propage vers le nœud dont l'évaluation de l'expression booléenne g est vraie.

$$\frac{\overline{l : decision(\langle g \rangle N_1, \langle \neg g \rangle N_2)}}{l : decision(\langle False \rangle N_1, \langle True \rangle \bar{N_2})} \text{ [decision1]}$$

$$\frac{N_1 \longrightarrow N_2}{\overline{l : decision(\langle g \rangle N_1, \langle \neg g \rangle N_2)} \longrightarrow \overline{l : decision(\langle g \rangle N'_1, \langle \neg g \rangle N'_2)}} \text{ [decision2]}$$

Ainsi, le scénario principal du XUC de la figure 5.5 peut être exprimé de la façon suivante: (pour simplifier l'écriture, les noeuds sont référencés par les noms de messages qu'il contient)

$$
\begin{aligned}
XUC_{initialization} &= S_0 \rightarrow l_1 : goInit \rightarrow N_1 \\
N_1 &= l_2 : Fork(l_3 : goInitAck \rightarrow N_2, l_4 : initConsole \rightarrow L_4) \\
N_2 &= l_5 : 2.Join(l_6 : addConsole) \rightarrow l_6 : addConsoleAck \rightarrow N_3 \\
N_3 &= l_7 : Final(success) \\
N_4 &= l_8 : Decision(\langle initConsoleAck \rangle \rightarrow l_5, \langle \neg(initConsoleAck) \rangle \rightarrow N_5 \\
N_5 &= l_9 : Final(failure))
\end{aligned}
$$

5.2.3 Sémantique des étapes d'un XUC

Le scénario d'un XUC décrit la manière avec laquelle les différentes étapes vont s'enchainer. La section précédente précise la sémantique de cet enchainement. Dans cette section, nous précisons la sémantique des étapes des XUC.

Nous avons opté pour la structuration des phrases pour exprimer les échanges entre le modèle et son contexte. Celle-ci est inspirée des travaux de H. Zhu *et al.* [Zhu & Jin 2002, Zhu *et al.* 2002]. Ces derniers ont proposé une approche pour tester statiquement des exigences logicielles définies sous forme d'un scénario de tâches (*task scenarios*). Nous reprenons cette idée pour décrire les étapes du scénario d'un XUC afin de faciliter la production de modèles de contextes précis et complets. La figure 5.6 présente la structuration des étapes telles qu'elles ont été proposées dans [Zhu *et al.* 2002].

Process P (*agent*) sends (*action*) data x of type real (*object*) to the terminator display (*receiver*) where $x > 0$. (*constraint*)

Figure 5.6 : Structure d'une étape selon [Zhu *et al.* 2002]

Ainsi, une étape d'un XUC est composée d'un agent qui sera dans le cas de la description du contexte soit un acteur de l'environnement ou un composant du système à vérifier.

Les actions possibles proposées par Zhu *et al.* sont:

recieves la réception d'une donnée d'un autre agent,

sends l'envoi d'une donnée vers un autre agent,

obtains l'obtention d'une donnée à partir d'un état interne du système,

updates la mise à jour de l'état du système

performs l'exécution d'un calcul par un composant du système.

Dans la mesure où les actions prises en compte dans CDL sont les **envois** et **réceptions** de messages, nous nous limitons, pour l'instant à ces deux types d'action dans les

étapes du scénario décrit dans le corps d'un XUC. D'autres actions telles que "*update*" et "*perform*" sont utilisables dans le scénario des XUC pour mettre à jour l'état du système pour déclencher un calcul, respectivement. Toutefois, les actions correspondantes ne sont pas prises en charge dans CDL pour le moment. Cela peut faire l'objet d'une extension du langage CDL.

Le champ *object* fait référence à une donnée ou une entité faisant partie du vocabulaire du domaine (c.-à-d. le nom d'un message). En plus du nom de la donnée, d'autres informations peuvent être incluses dans ce champ (telles que le type de la donnée, les arguments d'un message...).

À partir de cette structuration, une *étape* est formée en récupérant les informations relatives aux différents champs dwepuis les différents modèles de conception et/ou les dictionnaires de données du système étudié. Pour exprimer une étape, l'utilisateur s'appuie sur les termes référencés dans le vocabulaire du domaine (le(s) fichiers DSpec). En effet, dans l'approche proposée dans cette thèse, nous nous basons sur la définition d'une spécification de domaine (chapitre 4) pour représenter les différentes notions qui forment le vocabulaire du système étudier.

5.3 Sémantique des relations entre les XUC

UML2 [OMG 2007] propose deux types de relations entre les use cases:

1. les relations de généralisation/spécialisation permettant de préciser que des use cases sont des généralisations/spécialisations d'autres use cases.

2. les relations d'inclusion/extension permettant de spécifier des sous-ensembles d'interactions (comportement) sous forme d'un cas d'utilisation séparé. Ce use case peut alors être partagé en l'incluant dans plusieurs use cases qui partagent le comportement.

Nous avons fait le choix de n'utiliser que la relation d'inclusion entre les XUC. Ce choix est motivé par:

- la relation d'inclusion entre les XUC permettras la réutilisation des scénarios capturés par ces derniers et une meilleure organisation des cas d'utilisation,

- la sémantique de la relation de généralisation entre les cas d'utilisation n'est pas clairement définie par le standard UML et plusieurs travaux ont montrés les ambiguïtés posées par cette relation pour les cas d'utilisation [Cockburn 2000, Jac 2000, Stevens 2007].

Dans les sections qui suivent, nous allons discuter sur la problématique posée par la relation de généralisation ainsi que l'introduction de la relation d'inclusion au sein des cas d'utilisation étendus.

5.3.1 Problématique de la relation de généralisation entre les use cases

La sémantique de la relation d'extension telle qu'elle est proposée par UML2 reste vague, et plusieurs travaux ont soulevé le problème [Cockburn 2000, Jac 2000]. L'auteur dans [Stevens 2007] présente une discussion approfondie sur la problématique de la relation de généralisation entre les cas d'utilisations (fig. 5.7). Dans notre approche, nous nous limitons à la relation d'inclusion entre les XUC dans la mesure où elle permet une composition hiérarchique des use cases et leur réutilisation. L'exemple de la figure 5.7 est issu de [Stevens 2007].

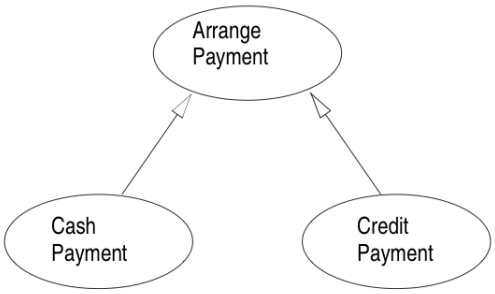

Figure 5.7 : Exemple de la généralisation entre use cases [Stevens 2007]

5.3.2 Sémantique de la relation d'inclusion entre les XUC

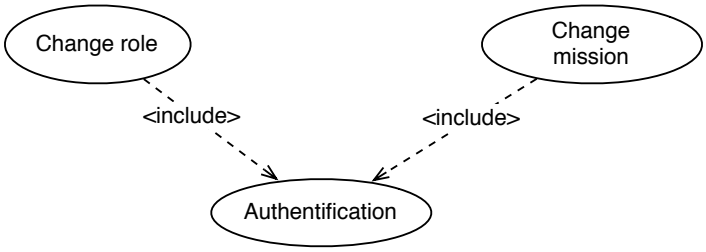

Figure 5.8 : Exemple de la relation include

Dans la section *semantics* relative aux use cases ([OMG 2007] page 593), UML2 précise que la relation d'inclusion entre deux use cases, (fig. 5.8), est définie comme suit:

"An include relationship between two use cases means that the behavior defined in the including use case is included in the behavior of the base use case. The include relationship is intended to be used when there are common parts of the behavior of two or more use cases. This common part is then extracted to a separate use case, to be included by all the base use cases having this part in common. ... Execution of the included use case is analogous to a subroutine call. All of the behavior of the included use case is executed

at a single location in the included use case before execution of the including use case is resumed." .

Ainsi, pour rester dans le même contexte sémantique proposé par le standard, nous définissons la relation d'inclusion entre les XUC comme un appel de fonction. De ce fait, le comportement du cas d'utilisation inclue est exécuté en totalité lorsqu'on atteint le point d'inclusion dans le use case de base. Pour cela, nous avons défini des points d'inclusions à l'intérieur des scénarios des cas d'utilisations étendus qui utilisent cette relation pour inclure le comportement d'autres XUC.

Définition 1 (Point d'inclusion) : Un point d'inclusion au sein d'un XUC (dit de base XUC_b) est le point à partir duquel le comportement du XUC inclus commence son exécution. A la fin de cette exécution, XUC_b reprend son exécution juste après ce point:

- Chaque point d'inclusion est relatif à un et un seul XUC_i tel que $XUC_i \neq Xuc_b$
- Un point d'inclusion peut se substituer à une étape s dans le scénario principale MS du XUC_b.
- Lorsqu'un XUC_b inclue un autre XUC_i à travers la relation $XUC_b \xrightarrow{I} XUC_i$, la transition t incidente vers le point d'inclusion substitue celle sortant du nœud de départ S_0 de XUC_i. De même, la transition ciblant le nœud final S_f de XUC_i cible le nœud incident du point d'inclusion de XUC_b.

5.4 Discussion et synthèse

Nous avons listé les problèmes qui limitent l'utilisation du langage CDL au sein des processus de vérification industriels dans la section 5.1.1. Notamment l'écart sémantique entre les modèles manipulés par les ingénieurs au cours des processus de développement et les modèles CDL. Puis nous avons défini un langage orienté utilisateur pour réduire ce gap.

Par la suite, nous avons introduit un langage pour la spécification des scénarios d'interaction entre le composant sous validation et son contexte. Ce langage s'appuie sur les cas d'utilisation UML et les étend par la définition des scénarios directement au sein de chaque cas d'utilisation. Les scénarios d'interaction sont définis sous la forme d'un enchaînement d'étapes exprimées à l'aide de phrases simples. Ces phrases référencent les éléments du domaine du système étudié afin de limiter le risque d'ambiguïté sur le vocabulaire utilisé.

Autre que le souci de simplicité et de cohérence des scénarios capturés par les cas d'utilisation étendus, nous avons ajouté un mécanisme d'exceptions/handlers afin de spécifier les scénarios d'exceptions en cohérences avec les scénarios nominaux. Ceci est particulièrement pertinent dans une configuration industrielle où la complexité du système en développement exige une description, la plus exhaustive possible, du comportement de

son environnement.

Nous avons détaillé les différentes constructions du langage ainsi que la sémantique des relations des XUC pour la spécification des modèles comportementaux des différents acteurs de l'environnement dès les premières phases du processus de développement. La sémantique des relations entre les étapes, présentée à la section 5.2.2, a été inspirée de celle proposée pour les diagrammes d'activités. L'objectif d'une telle sémantique est de pouvoir analyser automatiquement le comportement capturé par les scénarios des XUC afin de pouvoir extraire les comportements des acteurs de l'environnement. Ce comportement servira à la génération de modèles CDL en vue de la vérification formelle des exigences.
Les algorithmes de transformations de modèles XUC vers des spécifications CDL sont présentés dans le chapitre 7.

Nous présentons, dans le chapitre suivant, un langage dédié à la capture et la formalisation des exigences afin de pouvoir générer des propriétés CDL à partir des exigences manipulées par les utilisateurs.

6 Formalisation des exigences

Ce chapitre présente notre contribution pour réduire l'écart entre les exigences textuelles rédigées lors des premières phases du processus de développement et les exigences formelles utilisées pour la vérification formelle. En effet, dans un processus d'ingénierie, les exigences sont généralement traduites manuellement vers un formalisme disposant d'une sémantique formelle précise pour permettre leur vérification. Comme nous l'avons mentionné dans la section 2.3, de nombreux travaux ont proposé des approches facilitant l'expression des exigences telles que les patrons de spécification des propriétés ou les modèles d'exigences. Toutefois, en pratique, les exigences sont écrites sous forme textuelle dans des documents d'exigences ou, dans le meilleur des cas, dans des modèles d'exigences [Ryan 1992, Nuseibeh & Easterbrook 2000].

Dans ce chapitre nous reprenons l'approche basée sur l'utilisation des patrons de propriétés qui ont démontré leur utilité pour faciliter la formalisation des propriétés comportementales et temps réel. Nous proposons la définition d'un langage de spécification des exigences sous la forme d'un langage naturel contraint pour faire le lien entre les exigences textuelles et les patrons de propriétés CDL (présenté à la section 3.3). Nous appelons ce langage URM (pour *User Requirements Models*). Ce dernier exploite un vocabulaire du domaine du système étudié afin de garantir la cohérence des exigences formulées. Ce vocabulaire vise à préciser la signification des notions utilisées dans la spécification des exigences avec des liens vers le vocabulaire du domaine du système étudié (présenté dans le chapitre 4).

Dans un premier temps, nous rappelons brièvement l'intérêt des patrons de spécification et leur efficacité dans les processus de formalisation des exigences. Puis, nous présentons le principe de formalisation des propriétés à l'aide du langage CDL pour ensuite présenter notre langage de spécification des exigences et le processus proposé pour les formaliser.

6.1 La spécification de propriétés en contexte industriel

Dans cette section, nous présentons une synthèse des travaux existants dans l'état de l'art, présentés en chapitre 2, afin de positionner précisément notre contribution dans le cadre de la spécification et la formalisation des exigences.

6.1.1 Etat de l'art sur la formalisation des exigences

La difficulté liée aux systèmes logiciels actuels est que le nombre des exigences peut d'avérer très important. Ce qui rend difficile la tâche de s'assurer que celles-ci ne peuvent pas conduire à de multiples interprétations, qu'elles n'omettent pas de spécifier certains cas, ou qu'elles sont toutes cohérentes les unes avec les autres [Darimont & van Lamsweerde 1996]. D'où le besoin de pouvoir analyser ces exigences de façon automatisée, ou du moins assisté.

Pour faire face à ce problème, plusieurs travaux ont proposé des techniques permettant la vérification et l'analyse des exigences. Ces travaux peuvent être classés en deux grandes familles: soit rédiger les exigences directement sous forme de langages formels, soit l'utilisation de mécanismes de traduction du langage naturel vers des langages formels:

- Concernant la spécification directe en langage formel, le problème vient du fait qu'il n'est pas envisageable dans tous les contextes industriels de disposer de personnel compétent dans ces langages, et les experts métiers ne sont pas forcément des informaticiens [Cheng 2007].

- Les travaux qui proposent la traduction des exigences en langage naturel vers un langage formel sont multiples [Fantechi *et al.* 1994, Fuchs *et al.* 1999, Fabbrini *et al.* 2002]:

  Dans [Fantechi *et al.* 1994], les auteurs proposent une traduction automatique du langage naturel vers la logique temporelle ACTL. La traduction s'appuie sur un dictionnaire rempli par l'utilisateur.

  Ambriola *et. al* proposent dans [Ambriola & Gervasi 2006] un analyseur du langage naturel qui permet de produire différents types de modèles comme des modèles de flots de données [Ladkin & Leue 1995]. L'analyse des différents modèles permet alors de détecter un certain nombre d'anomalies dans les exigences: l'inconsistance des diagrammes de flots de données, l'ambiguïté, et la redondance.

  Fuchs *et. al* propose un langage naturel contrôlé pour remplacer la logique du premier ordre dans [Fuchs *et al.* 1999].

L'approche de formalisation des exigences proposée dans cette thèse entre dans le cadre de la deuxième catégorie. Nous proposons aux utilisateurs de rédiger leurs exigences sous une forme proche de celle utilisée dans les cahiers des charges, mais respectant une grammaire et s'appuyant sur un dictionnaire de données. Nous nous appuyons ensuite sur un système de patrons de propriétés afin de générer les spécification formelles exigées par les *model checker*.

Dans le cadre de notre approche, la grammaire du langage URM nous permet de faire une analyse structurelle sur les exigences rédigées afin d'identifier le(s) patron(s) de spécification de propriétés à appliquer pour formaliser chacune des exigences. Nous avons considéré les patrons de spécification de propriétés [Dwyer *et al.* 1999, Konrad & Cheng 2005a] sur lesquels le langage CDL s'est appuyé pour la formalisation des propriétés pour ensuite pouvoir effectuer la génération des automates observateurs correspondants.

6.1.2 Principe de formalisation de propriétés dans CDL

Dans le langage CDL, les exigences considérées correspondent aux patrons de spécification suivants:

Response Une ou plusieurs occurrences des événements répond à une ou plusieurs occurrences des événements.

Precedence Une ou plusieurs occurrences des événements est précédé par une ou plusieurs occurrences des événements

Absence Une ou plusieurs occurrences des événements n'arrive pas durant l'exécution.

Existence Une ou plusieurs occurrences des événements doit arriver durant l'exécution avant le délai spécifié.

Le but de notre approche est de pouvoir identifier le ou les patrons CDL adéquats à appliquer pour la formalisation des exigences. En effet, si les patrons de formalisation de propriétés CDL ont démontré leur efficacité lors de la formalisation des propriétés, le travail nécessaire, pour déterminer quel patron appliqué en fonction de l'exigence de départ, n'est pas trivial. En effet, à partir d'une exigence textuelle (comme celles présentées en annexe dans la section A.1), il faut arriver à déterminer le patron de propriété à appliquer afin de pouvoir générer l'automate observateur correspondant. De ce fait, nous présentons dans le reste de ce chapitre le langage URM qui nous permet d'analyser les exigences spécifiées pour pouvoir automatiser le processus d'identification des patrons de propriétés adéquats.

D'un autre côté, pour pouvoir lier les propriétés formalisées à des contextes spécifiques, celles-ci doivent prendre en compte l'état du système et/ou se rattacher à une exécution spécifique du contexte. Ainsi, il faut donc pouvoir observer en temps réel, c'est-à-dire durant l'exploration du modèle composé avec son contexte, l'état du système.

En effet, un automate observateur est traduit de l'expression d'une propriété. Cette expression référence un prédicat (appelé *scope*) dont la valeur dépend de l'état du système. Le *scope* de la propriété permet d'activer un observateur selon l'état du système. Ainsi, les propriétés exprimées doivent pouvoir référencer:

- un scope, c'est-à-dire le domaine d'exécution pour le lequel la propriété est active.
- des évènements dont l'occurrence est détectée lors de l'exploration.

Ainsi, pour observer l'état du système lors de l'exploration, un observateur spécifique (Observateur d'état) mémorise les états pertinents du système. Ces états peuvent être regroupés dans une liste de sous-états. Chaque sous-état est défini à partir de sous-états et/ou d'états élémentaires du système qu'OBP est capable d'évaluer. Aussi, les évènements sont déclarés pour faciliter l'écriture de la propriété avec le mot clé *event*. Ainsi, un état élémentaire du système est une expression de la forme:

- un processus P du système est dans l'état X (noté $state(P) = X$)

- une variable V a la valeur V1 (noté $eval(V) = V1$)

Lors de l'exploration du modèle par *OBP Explorer*, chaque exécution d'une transition d'un processus du système est suivie d'une mise à jour de l'état du système par l'observateur. C'est cet état qui est référencé sous forme de *scope* dans l'expression des propriétés. L'observateur mémorise l'ensemble des états, et les sous-états constitués, par une combinaison logique des états élémentaires du système. Chaque sous-état sera ensuite référencé dans une propriété par la déclaration d'un *scope*.

Pour une phase de vérification d'un ensemble des propriétés, OBP examine la spécification des propriétés et construit un espace d'états regroupant l'ensemble des sous-états référencés dans les propriétés via les *scopes*. Supposons que les 3 sous-états A, B et C soient référencés dans les propriétés par les *scopes* A, B et C. L'observateur d'état mis en œuvre lors de l'exploration évalue ces 3 sous-états.

Chaque scope doit être déclaré dans le programme CDL. Nous illustrons sur un exemple d'un *scope* A, correspondant à un sous-état du système, qui mémorise l'état d'une combinaison logique de variables a, b, c, d du système :


```
Scope A
{
        a and (b or c or (not d))
}
```

Le *scope* A prend les valeurs `"true"` ou `"false"` en fonction des valeurs de a, b, cet d. Les valeurs a, b, c et d sont évaluées après chaque exécution du système.

Exemple :

- $a : state(P) = X$
- $b : eval(V) = V1$

Ainsi, pour formaliser l'exigence suivante:

Si le processus P1 du système est dans l'état X
et si le processus P2 du système est dans l'état Y
et si le processus P3 n'est pas dans l'état Z
et si la variable V a pour valeur V1
et si la variable W n'a pas la valeur V2
alors l'occurrence de l'événement evt1 est suivie de l'occurrence de l'événement evt2

L'expression en CDL est donnée comme suit:

Code CDL

```
Property R
{
        scope A ;
        An exactly one occurrence of evt1
        leads-to
        An exactly one occurrence of evt2
}

Scope A
{
        state (P1) = X  and
        state (P2) = Y and
        (not state (P3) = Z) and
        and eval (V) = V1 and
        (not eval (W) = V2)
}
```

6.1.3 Vue globale de l'approche proposée

La figure 6.1 résume notre approche pour passer des exigences textuelles des cahiers des charges aux patrons de propriétés proposés par le langage CDL.

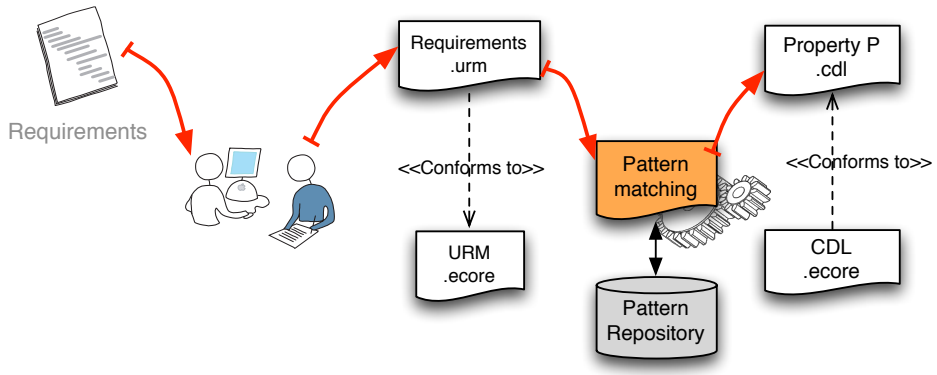

Figure 6.1 : Approche de génération de propriétés CDL à partir des exigences textuelles

Notre approche consiste à utiliser un DSL textuel permettant aux ingénieurs de spécifier des exigences analysables par des outils de façon qui se rapproche à la rédaction d'exigences textuelles qu'ils ont l'habitude de produire. En effet, nous avons montré que le principal inconvénient du langage naturel reste l'ambiguïté: une exigence écrite en langage naturel donne souvent lieu à de nombreuses interprétations. Or pour une vérification formelle, les exigences sont à la base des propriétés que l'on désire vérifier sur le modèle du système développé. Le danger est qu'une même exigence soit interprétée différemment de l'interprétation initiale qu'avait le rédacteur de l'exigence.

Pour conserver l'utilisation du langage naturel et limiter les dangers liés aux ambiguïtés, nous avons proposé un langage de spécification d'exigences assez simple, pour une utilisation facile en pratique, incluant les mécanismes permettant l'identification des

patrons de spécification d'exigences adaptées à chaque exigence exprimée.

La particularité du DSL proposé, que nous appelons URM pour *User Requirement Model*, est qu'il est basé sur l'utilisation du langage naturel utilisé par les différentes parties prenantes du projet tout en proposant les constructions du langage les plus utilisées. URM est basé sur une base de connaissance dans laquelle sont définis les différents mots clés spécifiques au système en développement. Celle-ci a pour but de forme un cadre commun avec les contextes exprimés dans le langage XUC pour l'exploitation des contextes lors de la vérification.

Ensuite, nous avons développé un algorithme (présenté dans la section 7.2) pour la détection automatique des patrons de propriétés à appliquer pour formaliser chacune des exigences exprimées dans URM. Cet algorithme utilise un référentiel regroupant les fragments de patrons de propriétés proposés dans la littérature. L'algorithme proposé identifie la structure de l'exigence utilisée en se basant sur un référentiel de fragments de patrons de propriétés et génère une formalisation de l'exigence directement dans la syntaxe CDL. De plus, cette étape permet de déceler les constructions structurelles non prévues, par le langage, dans les exigences formulées par l'utilisateur.

Dans la suite de cette section, nous allons présenter le langage URM et mettre en avant les particularités qui le distinguent des langages de spécification de propriétés existants. En suite, nous allons détailler les étapes permettant l'identification des patrons de propriétés à utiliser pour formaliser chacune des exigences du cahier des charges.

6.2 URM: Un langage de spécification des exigences

Dans cette section, nous allons commencer par présenter des exemples d'exigences telles que nous les trouvons dans les cahiers des charges industriels. Ces exigences sont issues du cas d'étude de l'*AFS*. Ensuite, nous présentons les différents concepts qui forment le langage URM permettant de reformuler les exigences présentées.

6.2.1 Exemple d'exigences industrielles

Les exigences sont présentées ici pour illustrer les concepts du langage que nous proposons. Dans le chapitre 9, dédié à l'étude de cas, nous traitons un ensemble de 12 exigences, toutes issues du cahier des charges du cas d'étude *AFS*.

> [**Exigence AFS-SM-020**] If the *SM* is in state Standby, and the mission is not yet selected, upon reception of a valid *EvtRequestLoginHmi* message from the *HMI*, the *SM* shall send the following messages to the HMI in this order :
> (1) *EvtAckHmi* with result to TRUE, (2) *EvtCurrentUserHmi* with *Username* to *currentUsername*.

À partir de cette exigence, nous pourrons déjà dresser une liste de points importants qu'il faut tenir compte lors de la définition du langage URM pour la description d'exigences. Celui-ci a été proposé pour être à la fois proche de la compréhension des experts du domaine et qui leur permettre l'application des outils de traitement automatique des exigences:

1. Une exigences est formée d'une ou plusieurs phrases
2. Différents types de phrases peuvent être utilisées (simple, conditionnelles, modal)
3. Les éléments constituants les phrases font partie du vocabulaire du domaine : HMI, DP, *EvtAckHmi*, *ExternalMassStorage*, *Active_State* ...
4. Les exigences peuvent avoir des prés et des post conditions
5. Une exigence est définie dans un contexte particulier. Autrement dit, elle doit être vérifiée sous des conditions particulières.

Ainsi si on analyse d'exigence **AFS-SM-020**, nous pouvons identifier la structure suivante:

- Précondition : "If the *SM* is in state Standby and the mission is not yet selected"
- Phrase simple : "upon reception of a valid *EvtRequestLoginHmi* message from the *HMI*"
- Phrase modale: "the *SM* shall send the following messages to the HMI in this order : ..."

Ces deux phrases forment ce qu'on appelle une phrase conditionnelle.

Dans la section suivante, nous présentons la formalisation des concepts identifiés pour former notre langage de spécification des exigences.

6.2.2 Vue d'ensemble du langage

Dans le langage URM, une exigence est représentée par une ou plusieurs assertions (*statement*). Ces assertions référencent des éléments de la spécification du domaine tels que les acteurs du contexte, les composants du système, les noms et les arguments des messages échangés, les variables, etc. Chaque assertion exprimant une exigence est décrite sous une forme grammaticale bien précise. Le langage propose trois types d'assertions:

Les phrases simples de type sujet-verbe-complément

Les phrases modales pour décrire des modalités sur les actions du sujet

Les phrases conditionnelles pour exprimer des conditions

La composition de ces trois type de phrases sont définies dans le package *Phrases* du métamodèle URM. En effet, ce dernier est organisé en quatre packages. La figure 6.2 présente la structure du métamodèle du langage URM.

- *ConstrainedLanguage*: Définit les concepts nécessaires à l'expression des exigences à l'aide du langage naturel contrôlé. Le contenu de celui-ci est détaillé dans la section 6.2.3.1.

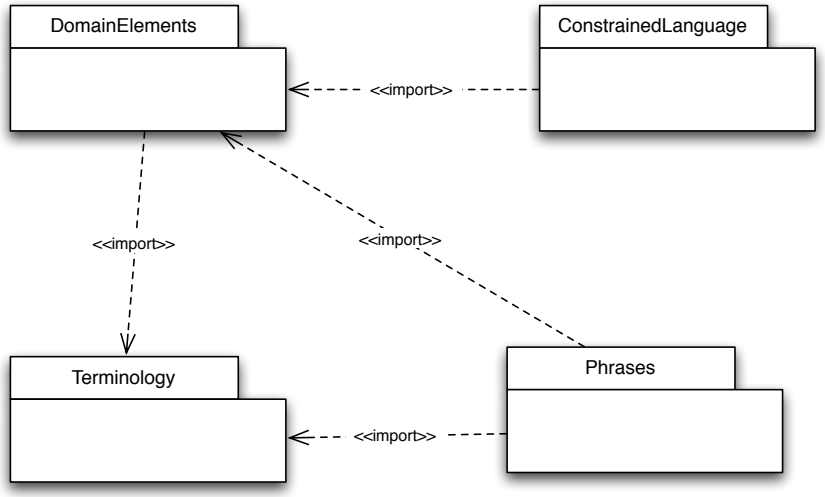

Figure 6.2 : Organisation du métamodèle URM en packages

- *DomainElements*: Regroupe les éléments du domaine référencé par les exigences URM. Le contenu de ce package ainsi que les explications relatives à la spécification du domaine sont présentés dans le chapitre 4 page 51.
- *Terminology*: Définit les différentes parties du langage qui composent les différents types de phrases du langage naturel contrôlé. (Voir la section 4.2.2 page 58).
- *Phrases*: Définit les différents types de constructions grammaticales autorisées dans le langage URM.

6.2.3 Description du métamodèle et sémantique

Dans le langage URM, les exigences sont décrites au sein d'une *spécification*. Celle-ci est l'élément de haut niveau qui contient la définition de l'ensemble des exigences d'un système particulier. Une spécification englobe un ensemble d'exigences qui sont décrites dans le package *Constrained Language*. Dans ce qui suit, on décrit le métamodèle du langage URM et on commence par le package de description du langage naturel contraint (*Constrained Languge*) qui représente l'élément central du métamodèle.

6.2.3.1 Constrained Language Package

Ce package contient les concepts et les constructions grammaticales propres au langage URM pour exprimer les exigences textuelles. Ces concepts sont représentés par la figure 6.3 qui représente un métamodèle conforme au métamodèle MOF.

Dans la suite de cette section, on donne une description détaillée de chaque métaclasse du métamodèle. Pour chaque métaclasse, nous présentons sa description, ses attribues, ses associations ainsi que ses généralisations directes et contraintes s'il y a lieu. Les métaclasses sont présenter selon leur importance et non selon l'ordre alphabétique.

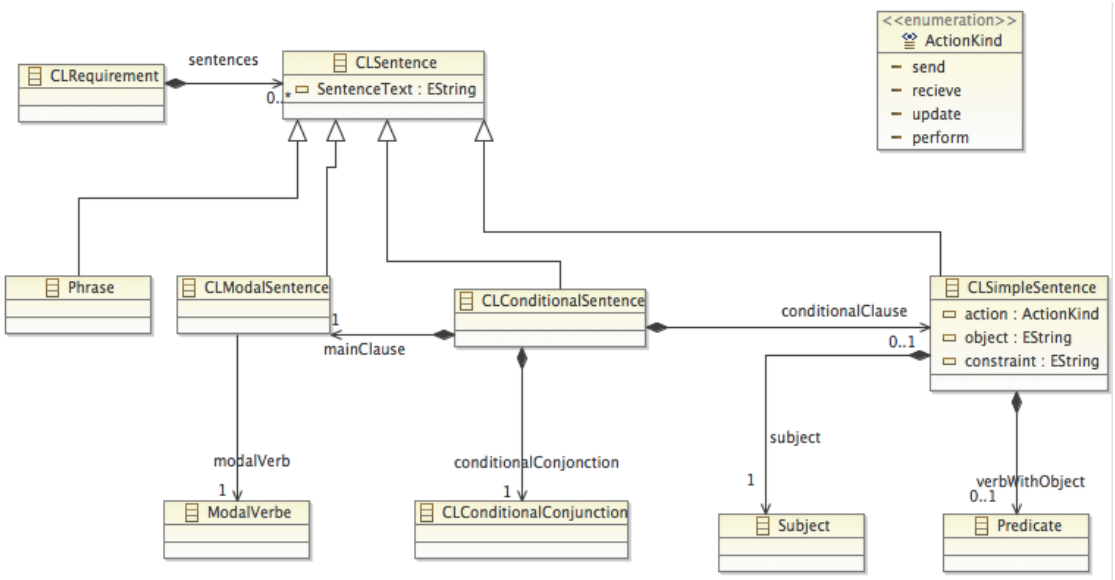

Figure 6.3 : Présentation du package *ConstrainedLanguage*

CLRequirement Une exigences est formée d'une ou plusieurs phrases (*sentences*) de type *CLSentence*.

CLSentence est une classe abstraite représentant une phrase qui peut se spécialiser en différents types de constructions grammaticales. L'attribue *SentenceText* représente cette phrase sous forme d'une chaîne de caractères libres (sans aucune contrainte grammaticale). Celui-ci est utilisé pour les constructions de phrases qui ne sont pas encore prises en charge par le langage.

CLSimpleSentence Représente une phrase simple sous la forme: sujet, verbe complément (SVO: *Subject, Verb, Object* en anglais). La partie VO est représentée par un prédicat *Predicate* qui pointe vers une phrase verbale (*Phrase::VerbPhrase*). La classe *CLSimpleSentence* dispose de trois attributs:

- *action*: représente l'action que produit le sujet (*Subject*) sur un objet particulier du système.
- *object*: représente un objet du système sur lequel est produite l'action.
- *constraint*: est un attribut facultatif qui représente une contrainte sur l'objet de la phrase.

CLModalSentence Représente une phrase avec un verbe modal (*ModalVerb*). A la différence des phrases simples (*CLSimpleSentence*), les phrases modales expriment soit (1) la priorité d'une activité, soit (2) la modalité, soit (3) l'obligation ou la possibilité pour le sujet d'exécuter l'action.

CLConditionalSentence Représente une phrase avec une condition (décrite par une *conditionalClause*) commençant par une conjonction conditionnelle (*conditionalConjonction*) et se termine par une *mainClause* qui représente la conséquence de si la

condition s'avère vraie. La conjonction conditionnelle est représentée par *CLConditionnalConjonction* telle que "if" par exemple. La condition est une phrase simple de type *CLSimpleSentence*. La conséquence de la condition est une phrase simple (*CLSimpleSentence*) ou modale *CLModalSentence*.

Subject Cette classe représente le sujet d'une phrase *CLSimpleSentence* d'un point de vue grammatical. Le sujet fait référence à un acteur où un composant du système étudié de la description du domaine (voir le chapitre 4 décrivant la spécification du domaine). Cet élément peut exécuter l'action décrite dans la phrase simple (*CLSimpleSentence*).

6.2.3.2 Phrases Package

Ce package contient les entités du langage permettant la formulation des phrases dans un langage structuré. Ces phrases représentent des termes (les noms) associés avec d'autres termes (verbes et adjectifs). Une phrase est toujours présentée dans le contexte d'un nom (*Subject*) et est potentiellement associée à un qualificatif ou un modifiant. Un autre type de phrases sont les phrases verbales qui décrivent le contexte d'un verbe. La figure 6.4 montre l'organisation du package *Phrases*.

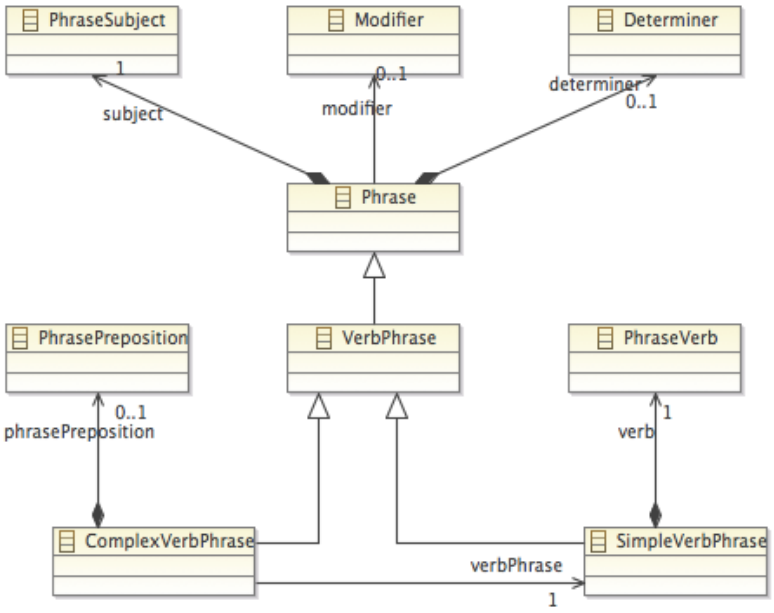

Figure 6.4 : Organisation du package *PhrasesPackage*

Phrase: Une phrase est de type *ConstrainedLanguage::CLSentence*. Une phrase contient un *objet* de type *Terminology::Noun* et potentiellement des compléments d'objet de type *Terminology::Modifier* et *Terminology::Qualifier*. L'objet contenu dans une phrase représente le *nom* d'un élément du domaine (*DomainElements::DomainElement*).
Exemples: "*Console*", "Logged *console*"

Object est le sujet de la phrase. Il est de type *Terminology::Noun*

VerbPhrase Une phrase verbale décrit une opération qui peut être exécuté en association avec un *Objet* de type *Terminology::Noun*.

SimpleVerbPhrase décrit une phrase de type Verbe-Complément qui peut être utilisé en complément d'un sujet.
Exemples: "send request"

ComplexVerbPhrase décrit une relation entre deux objets. La construction grammaticale d'un phrase verbale complexe est de type (Verbe - Complément - Complément)
Exemples: "send request to *SystemManager*"

Modifier Indique comment l'objet en question doit être interprété dans le contexte de la phrase. De cette façon, de sens de l'objet est distingué de celui décrit par la terminologie.

Qualifier qualifie le sujet de la phrase pour décrire son contexte dans la phrase en terme de quantité et de variabilité.

PhraseVerb représente le verbe de la phrase. Il est utiliser dans les *SimpleVerbPhrase* et *ComplexVerbPhrase*. Un verbe est de type *Terminology::Preposition*.

6.3 Discussion et synthèse

Dans ce chapitre, nous avons présenté un langage de spécification des exigences afin de pouvoir les vérifier à l'aide des outils de vérification formelle existant dans un contexte industriel. L'objectif est d'obtenir une spécification suffisamment précise pour être mise en avale d'une chaîne de transformations de modèles. Celle-ci permettras l'obtention automatiquement des artefacts exploitables par ces outils.

Dans cet esprit, nous avons proposé un langage de spécification d'exigences, appelé URM pour la rédaction des exigences du client afin de permettre la génération automatique des propriétés CDL tout en facilitant les communications et les discussions sur les exigences. URM s'appuie sur une spécification du domaine sous la forme d'une base de connaissance du domaine listant l'ensemble des entités du domaine.

Ainsi, avec la formalisation des exigences présentée dans ce chapitre et la formalisation des contextes présentée dans le chapitre 5, nous arrivons à la production de l'ensemble des artefacts nous permettant l'application des techniques de vérification formelle par exploitation des contextes. Autrement dit, nous arrivons au point où on peut obtenir des modèles CDL dans lesquels sont spécifiés les comportements des acteurs du contexte et les propriétés à vérifier. Dans les chapitres suivants, nous présentons notre méthodologie de vérification et nous expliquons comment les différents concepts présentés jusqu'à présent vont s'intégrer au processus de développement.

TRANSFORMATION DE MODÈLES ET MÉTHODOLOGIE

TELECOM Bretagne

7 Génération de modèles CDL

Dans la partie précédente, nous avons présenté trois langages, dits orientés utilisateur, pour faciliter les activités de spécification et de formalisation des exigences et des contextes. Nous présentons dans ce chapitre les transformations de modèles permettant la génération des modèles CDL pour la chaine d'outils OBP présentée dans la section 3.4.

La figure 7.1 illustre la chaine de transformations proposée. Le processus de déroule en deux temps: Dans une première partie, on génère la description détaillée du comportement des acteurs du contexte à partir des modèles XUC produits manuellement par l'utilisateur. Dans une deuxième partie, nous procédons à la génération de propriétés formalisées CDL à partir des exigences URM. Les exigences du cahier des charges sont réécrites manuellement par l'utilisateur dans la syntaxe URM. En suite un processus automatique permet d'analyser les exigences URM pour identifier les patrons CDL adéquats à appliquer afin de générer le code CDL de la propriété. Les modèles XUC et URM référencent les éléments du domaine stockés dans la spécification du domaine sous la forme de fichiers Dpec.

7.1 Génération de contextes CDL à partir des modèles XUC

Afin de capturer les interactions entre le système en développement et son environnement, nous avons proposé dans le chapitre 5 le langage XUC. Ce langage constitue une extension des cas d'utilisation UML avec la spécification des différents scénarios d'interactions des acteurs directement dans le corps des cas d'utilisation. Aussi, les modèles XUC référencent les éléments du domaine capturés dans les fichiers DSpec pour améliorer la cohérence des modèles construits, mais aussi pour faciliter les traitements automatisés des modèles XUC.

Nous présentons dans cette section les différentes étapes permettant la génération de la partie contexte des modèles CDL destinés à être exploités par la chaine d'outils OBP. D'abord, nous rappelons la sémantique des liens vers la spécification du domaine (fichiers *.dspec*). Ensuite, nous détaillons les algorithmes de transformation permettant la traduction du comportement capturé par les modèles XUC sous la forme d'interactions CDL. Enfin, nous discutons sur la validité de cette chaine de transformations et des extensions possibles.

7.1.1 Liens avec la spécification du domaine

Au sein des étapes d'un XUC, les différents champs font directement référence aux entités du domaine stockées dans les modèles DSpec. Le langage DSpec (présenté dans le

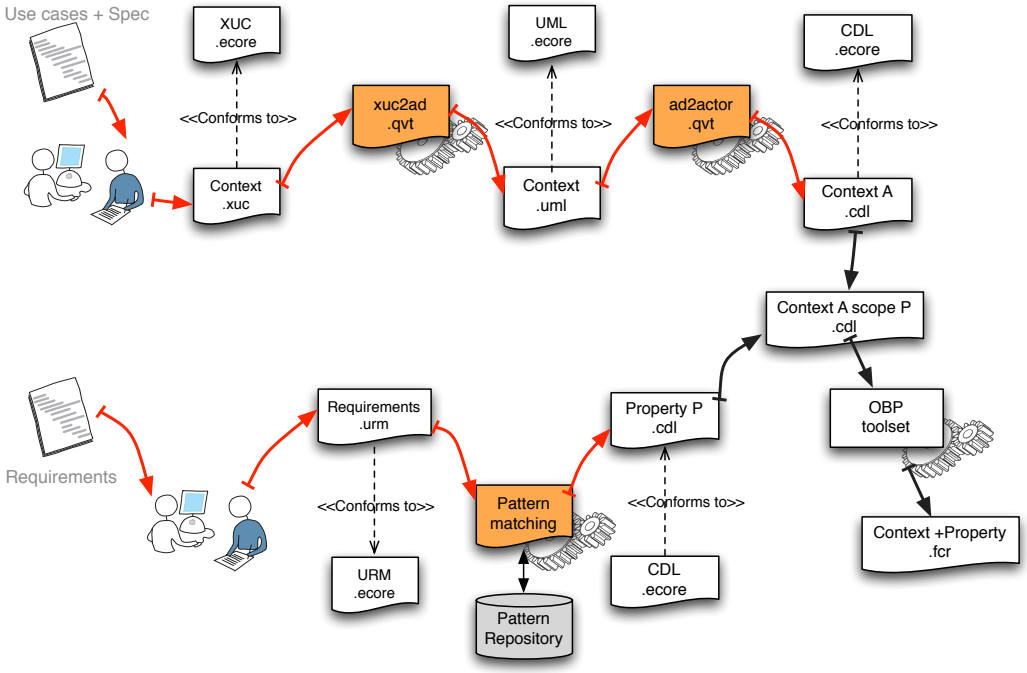

Figure 7.1 : Processus de génération de modèles CDL

chapitre 4) a pour but de lister et définir les différentes notions du domaine pour faciliter la spécification des scénarios des XUC mais aussi les exigences. Ce lien, entre les modèles DSpec et les modèles XUC, permet de déterminer la nature de chaque entité (acteur, composant du système, message, événement, argument de message). Cette information est exploitée par l'algorithme de traduction des XUC pour la construction de modèles CDL corrects. La figure 7.2 schématise les liens entre les fichiers XUC et les fichiers DSpec.

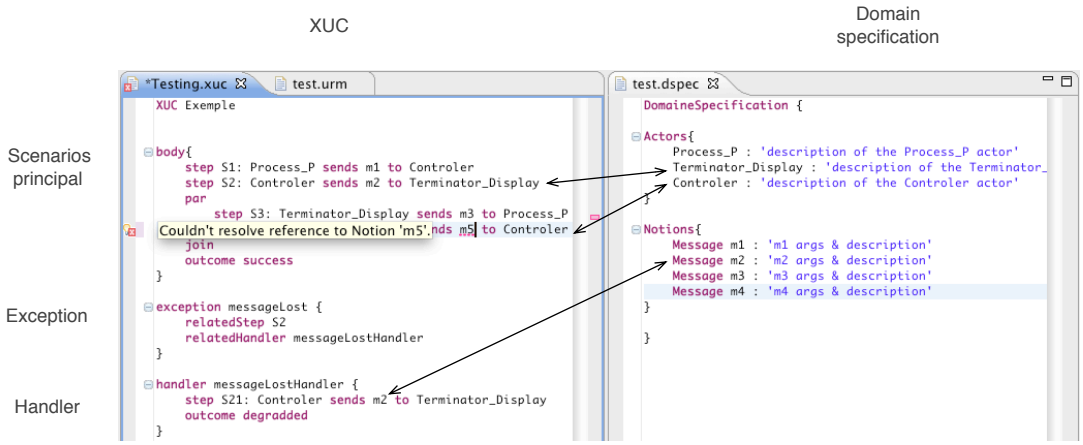

Figure 7.2 : Schématisation des liens entre fichiers XUC et la spécification du domaine

7.1.2 Génération de modèles de contextes

La génération de modèles de contexte consiste à transformer les scénarios capturés par les cas d'utilisation étendus en modèles CDL. Cette transformation se fait en deux temps; (1) d'abord un diagramme d'activité est généré exprimant les scénarios du cas d'utilisation, ensuite (2) applique un algorithme pour séparer les activités relatives à chacun des acteurs participant à ce cas d'utilisation. La figure 7.3 illustre ce processus de génération.

7.1.2.1 1˚ Transformation: Génération d'un diagramme d'activité pour chaque XUC

La traduction des cas d'utilisation XUC vers des représentations en diagrammes d'activités UML est inspirée des travaux de [Gutiérrez *et al.* 2008]. L'algorithme de transformation comporte les étapes suivantes:

1. Créer un noeud *activity* pour chaque XUC
2. Créer une partition d'activité (*activity partition*) pour chaque acteur participant au cas d'utilisation et une activité partition pour chaque composant du système impliqué dans les interactions
3. Créer une action (*action*) pour chaque étape (*Step*)
4. Ajouter les actions générées aux partitions d'activités de l'acteur identifié par l'attribut *performedBy* de l'étape.

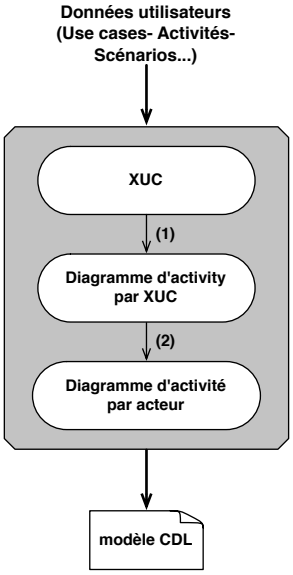

Figure 7.3 : Processus de génération d'un modèle CDL à partir d'un XUC

5. Créer un noeud décision (*decisionNode*) pour chaque exception et une nouvelle *activity* pour chaque *handler*.

6. Créer un noeud final pour chaque fin de scénario (identifié par le mot clé *outcome*) et le labelliser par le stéréotype correspondant à l'attribut *outputKind*.

7. Lier l'ensemble des éléments générés en utilisation les flots de contrôle du XUC en utilisant l'algorithme présenté ci-dessous.

Ainsi, chaque exception, identifiée dans un cas d'utilisation, est représentée par un noeud décision (*DecisionNode*) dans le diagramme d'activité généré. De plus, une activité représentant le *handler* de l'exception est aussi générée. Dans le cas où le *handler* n'est pas défini pour l'exception en question, un noeud final est généré dont le stéréotype correspond à l'attribut *OutputKind* de l'exception.

Une fois tous les éléments générés, l'algorithme de contrôle du flux prend le relais pour lier les noeuds du diagramme d'activités généré:

1. Faire le lien entre le noeud initial et la première action et entre le noeud final et la dernière action de l'activité principale;

2. Dans le scénario principal, chaque action est liée avec l'action correspondant à l'action référencée par la transition sortante *outgoing* (voir le métamodèle de la figure 5.4 à la page 69) si aucune exception n'est définit pour l'action en question;

3. Si une exception est définie pour l'action, créer une transition (*Control Flow*) entre l'action et le noeud décision correspondant à l'exception rencontrée;

4. Si plusieurs exceptions (respectivement *handler*) sont définies pour une étape, des transitions sont créées entre les noeuds décision (respectivement noeuds activités) et les actions correspondantes à l'étape;

5. Les noeuds décision créés sont reliés aux actions issues des étapes référencées par les transitions sortantes des étapes qui déclenchent l'exception.

7.1.2.2 2˚ transformation: Génération d'un diagramme d'activité pour chaque acteur de l'environnement

Un modèle CDL capture les interactions du contexte sous la forme de modèles comportementaux des différents acteurs avec lesquels le système interagit. Ainsi, pour pouvoir générer des modèles CDL, nous avons besoin de ces modèles comportementaux. À l'issue de la première transformation, nous obtenons un diagramme d'activités représentant les échanges du système avec tous les acteurs participant à ce cas d'utilisation. Nous avons proposé un algorithme permettant la séparation des comportements des acteurs dans des diagrammes d'activités séparés tout en préservant le comportement global du use case.

Pour expliquer l'algorithme en question nous utilisons une définition partielle d'un diagramme d'activité. Cette définition est suffisante pour la description de l'algorithme qui extrait les comportements des acteurs depuis le diagramme d'activité globale.

> **Définition 1 (Un diagramme d'activité) :** est un tuple $AD = \langle A, C, G, E, \Delta \rangle$ avec A l'ensemble des noeuds *activity*, C l'ensemble des noeuds de contrôle, E l'ensemble des transitions, G l'ensemble des partitions d'activités avec $A \cup C \cup E \subset G$. Δ définit la relation du contrôle de flux tel que: $\Delta : (A \cup C) \times A \to E$.

Ainsi, le principe de l'algorithme proposé est le suivant: Il commence par identifier les différentes partitions d'activité du diagramme d'activité globale. Chaque partition représente un acteur et regroupe l'ensemble de ses activités. Pour chaque partition, un nouveau diagramme d'activité est généré auquel est ajouté un noeud initial. ensuite, les activités de l'acteur sont parcourues dans le diagramme global pour les ajouter dans le nouveau diagramme d'activité. Et termine par la création du noeud final.

Selon le standard UML2 [OMG 2007], chaque élément des ensembles A, C et E peut être associé à une ou plusieurs partitions. Dans notre approche, nous considérons que les activités et les noeuds de contrôle sont contenus dans une seule partition à la fois puisque les acteurs de l'environnement ne partagent pas leurs actions dans le diagramme d'activité généré depuis le cas d'utilisation étendu. En effet, une action dans le diagramme d'activité globale correspond à une étape du XUC. Cette étape est décrite à l'aide d'une phrase simple (*CLSimpleSentence*) avec un seul sujet par phrase. De ce fait, les activités générées dans le diagramme d'activité globale correspondent à un et un seul acteur. ¨Par conséquent, on peut affirmer que, dans le cas de notre transformation, une activité n'est contenue que dans une partition d'activités à la fois.

Toutefois, il se peut qu'une transition lie deux activités appartenant à deux partitions

d'activités différentes. C'est le cas fréquent où deux actions successives sont exécutées par deux acteurs différents. Nous appelons ces transitions des *transitions traversantes* (*crossing edges*). Nous considérons que ces transitions font partie de la partition contenant l'activité source de la transition (identifiée par l'attribut *source*).

Ainsi, l'algorithme est présenté comme suit:

```
mapActivityGroup2ActivityDiagram() {
        ADi = createInitialNode()
        processOwnedElements(ADi)
        createActivityFinalNode(ADi)
}
```

Création du noeud initial:
Dans le diagramme d'activité global, une seule partition d'activité contient le noeud initial. Elle correspond à l'acteur qui déclenche le premier événement du XUC. Pour les autres partitions, la règle consiste à créer un nouveau noeud initial. Ainsi, pour une partition notée Gi, les noeuds initiaux sont créés en utilisant l'algorithme 1.


```
1:  if Gi contains an initial node then
2:      In ← Gi.getInitialNode
3:      Ei ← In.getOutgoingEdge
4:  else
5:      In ← new InitialNode
6:      Ei ← new Edge
7:  end if
8:  ADi ← new ActivityDiagram
9:  add In and Ei to ADi
10: return ADi
```

Algorithm 1: CreateInitialNode()

Traitement des noeuds de la partition d'activités:
Pour chaque partition d'activités, l'agorithme parcourt l'arborescence des éléments contenus pour les ajouter au diagramme d'activité de l'acteur correspondant. Les éléments du modèle concerné par ce processus sont les noeuds d'activité (A), les noeuds de contrôle (C) et les transitions (E). On considère que les transitions sont contenues par la partition contenant le noeud source de la transition. L'algorithme 2 illustre ce processus.

Traitement des transitions traversantes:
Pour les transitions qui lient deux noeuds appartenant à deux partitions différentes, l'algorithme consiste à créer une instance de la classe *AcceptEventAction* du métamodèle UML pour pouvoir synchroniser les deux diagrammes d'activités générés. L'instance créée représente un noeud de réception d'un signal arrivant de l'extérieur du diagramme d'activité. Ainsi, l'exécution de chaque action produit un signal (*SendSignalAction*) qui

```
Require: ActivityDiagram : ADi
n ← getRootElement(Gi)
e ← Gi.getFirstNode
if n isInstanceof InitialNode then
    ADD e to ADi
    LINK e WITH n.getOutgoingEdge.getTarget
else
    for all ed ∈ n.getIncomingEdge do
        processCrossingEdges(ed, ADi)
    end for
end if
for all element e ∈ Gi do
    if e isInstanceof Edge then
        if e.getTarget ∉ Gi then
            ed ← Gi.getNextNode.getIncomingEdge
            processCrossingEdges(ed, ADi)
            LINK e WITH EventNode resulting from ed
            ADD e to ADi
        end if
    else
        ADD e TO ADi
        nd ← e.getTarget
        ADD nd to ADi
        LINK e with nd
    end if
end for
```

ALGORITHM 2: ProcessOwnedElements(ADi)

```
Require: Edge : ed and ActivityDiagram : ADi
a ← ed.getSource
aea ← new AcceptEventAction
aea.setName(a.getGroupName + a.getName)
ADD aea to ADi
CREATE Edge between aea and ed.getTarget
```

ALGORITHM 3: ProcessCrossingEdges(ed, ADi)

sera intercepté par le noeud créé pour notifier le diagramme d'activité concerné.

Création des noeuds finaux:
Dans cette étape, l'algorithme crée, pour chaque diagramme d'activité généré, un ou plusieurs noeuds finaux. Les activités qui ne contiennent aucun noeud final terminent leurs comportements par une transition traversante (car le noeud final est contenu par une autre partition). Dans le cas où la dernière transition cible un noeud final, on crée un nouveau noeud final dans le diagramme d'activité concerné et il est labellisé par la valeur de l'attribut *outComeKind*. Si la dernière transition cible une action dans une partition différente, l'acteur termine sont comportement en succès.

Require: ActivityDiagram : ADi
1: ed ← Gi.getLastEdge
2: afn ← new ActivityFinalNode
3: **if** ed.getTarget isInstanceof ActivityFianlNode **then**
4: afn.setStereotype(ed.getTarget.getStereotype)
5: **else**
6: afn.setStereotype(Success)
7: **end if**
8: ADD afn to ADi
9: ed.setTarget(afn)

ALGORITHM 4: CreateActivityFinaleNode(ADi)

7.2 Génération des propriétés CDL à partir des exigences URM

Dans la section précédente, nous avons présenté l'algorithme de génération de la partie contexte du modèle CDL à partir des modèles XUC. Pour compléter le modèle CDL, cette section présente le processus de formalisation des exigences URM pour produire des propriétés CDL.

La figure 7.4 schématise le processus global de formalisation des exigences à l'aide du langage URM et des patrons de spécification de propriétés. Nous allons détailler ce processus et l'illustrer à l'aide d'exigences issues des cahiers de charges du cas d'étude *AFS* présenté dans la section 6.2.1.

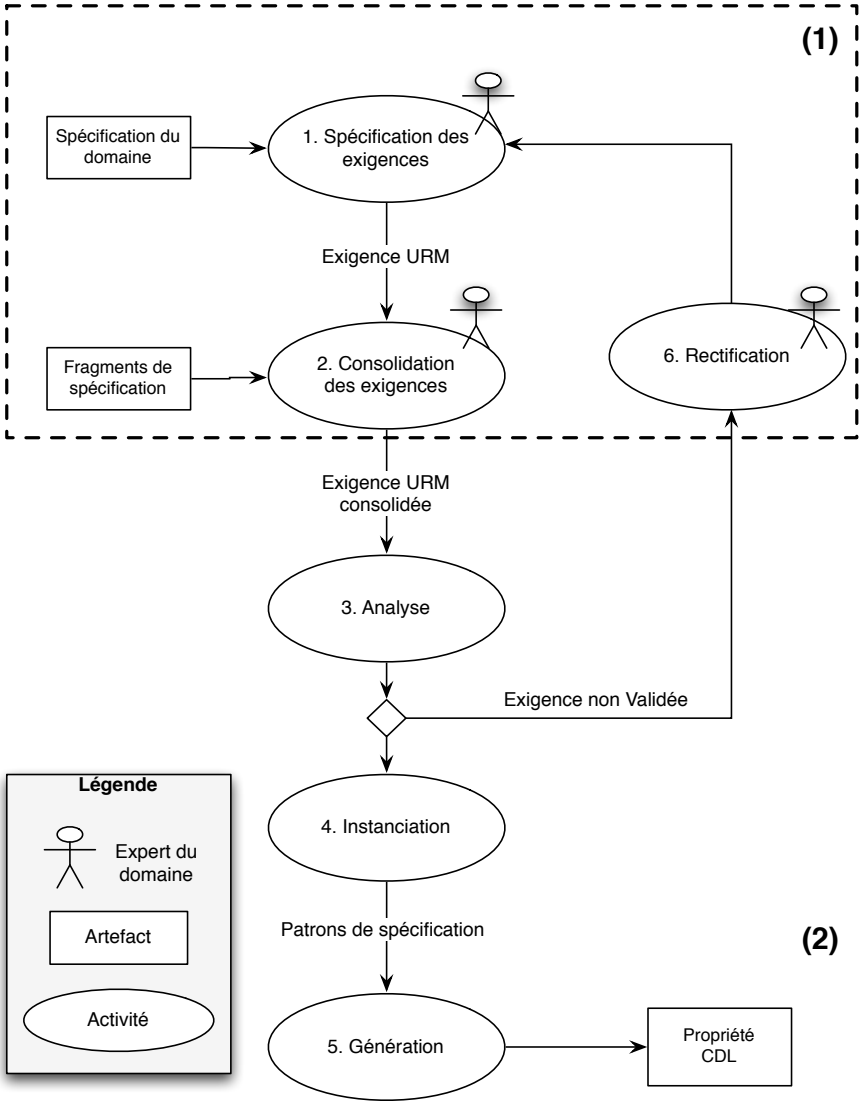

Figure 7.4 : Processus d'identification des patrons de propriétés dans les exigences

Pour clarifier l'application de ce processus, nous allons le découper en deux parties: (1) Une partie dite manuelle, où l'expert du domaine est en charge de produire les modèles de sortie de l'activité. Cette partie, englobe les activités de spécification et de consolidation des exigences de la figure 7.4 ainsi que l'activité de rectification. (2) Une partie de génération automatique qui concerne l'analyse et la génération des propriétés CDL.

7.2.1 Spécification des exigences

Dans notre méthodologie, nous avons proposé un langage de spécification des exigences s'appuyant sur du langage naturel contrôlé avec des construction grammaticales simples. Ceci dans le but de permettre aux experts métier utiliser directement le langage URM pour réécrire les exigences fournies dans le cahier des charges. Au cours de cette activité, celui-ci utilise les différentes constructions de phrases proposées par le langage et fait référence aux entités du domaine définies dans la spécification du domaine (un ou plusieurs fichiers *.dspec*). Cette spécification peut être complétée pour y ajouter les entités référencées par les exigences. A l'issue de cette étape, l'exigence de départ est réécrite en langage URM. La figure 7.5 illustre ce processus.

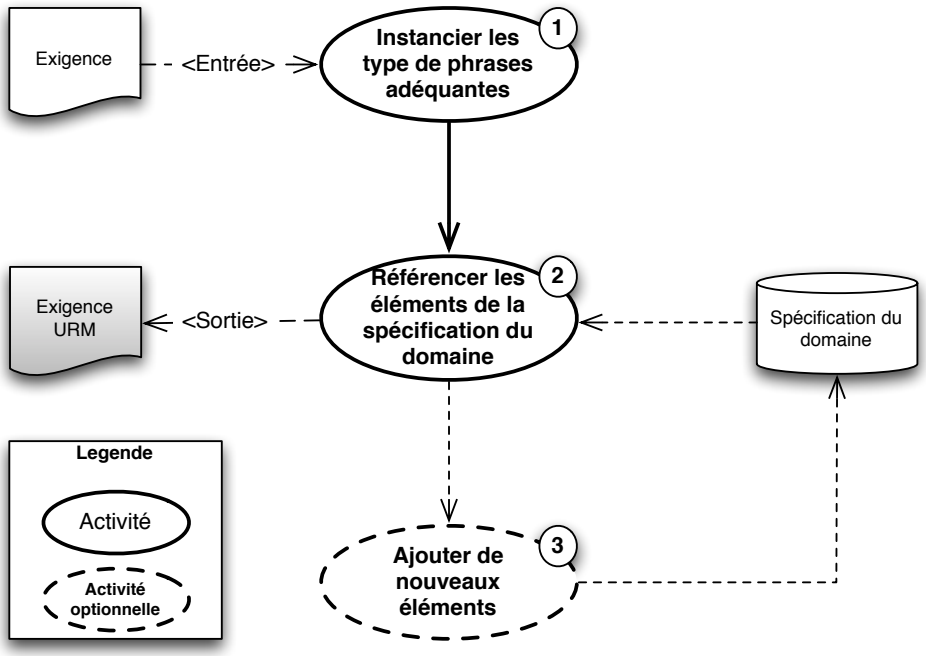

Figure 7.5 : Processus de spécification des exigences URM

Exemple:

Si on considère l'exigence **AFS-SM-020**, celle-ci est présentée comme suit dans la documentation:

If the SM *is in state Standby, and the mission is not yet selected, upon reception of a valid* EvtRequestLoginHmi *message from the* HMI*, the* SM *shall send the following messages to the HMI in this order :*
(1) EvtAckHmi *with result to TRUE, (2)* EvtCurrentUserHmi *with* Username *to* currentUsername*,*

L'activité de spécification consiste à identifier les types de phrases utilisés pour spécifier l'exigence ainsi que les éléments de la spécification du domaine référencé. Dans ce cas, l'exigence est formée d'une phrase conditionnelle "*CLConditionalSentence*":

if *CLSimplePhrase*, **Then** *CLModalPhrase*

La partie condition est formée de la concaténation de trois phrases simples :

- *"the* SM *is in standby"*
- *"the mission is not selected"*
- *"upon reception of..."*

La phrase modale est formée d'une succession d'action que le *SM* doit effectuer dans l'ordre:

- *"the* SM *shall send the following messages to the HMI in this order..."*

Le but de cette identification est d'utiliser les mots clés adéquats pour la réécriture, afin que l'exigence soit conforme au métamodèle du langage URM. De ce fait, le raffinement qui a été appliqué sur l'exigences **AFS-SM-020** est comme suit :

1. Identification de la structure de l'exigence et instanciation de deux phrases dans l'éditeur URM
2. Référencer les messages échangés de la spécification du domaine
3. Utilisation de la syntaxe URM pour l'affectation des valeurs aux arguments:
("EvtAckHmi with result to TRUE" devient "EvtAckHmi **with** result = TRUE")

Aussi, la phase de réécriture permet de s'assurer que tous les éléments référencés par l'exigence sont bien définis dans la spécification du domaine.
Ainsi, l'exigence **AFS-SM-020** est réécrite comme suit :

If *"the* SM *is in state Standby"*
and *"the mission is not yet selected"*
upon reception of *"a valid* EvtRequestLoginHmi *message from the* HMI*"*
then *the* SM ***shall send*** *the following messages to the HMI in this order :*
(1) EvtAckHmi ***with*** *result = TRUE, (2)* EvtCurrentUserHmi ***with*** Username = currentUsername*,*

Cette activité manuelle, bien qu'elle soie contraignante voire difficile à mettre en oeuvre pour un grand nombre d'exigences, est facilitée par l'éditeur d'exigence URM que nous avons implémenté dans le cadre de cette thèse. En effet, lors de cette activité, l'utilisateur a la possibilité de choisir, depuis une liste déroulante, le type de phrase adéquat pour l'exigence étudiée à partir d'un menu, puis de remplir les différents champs en sélectionnant les éléments du domaine stockés dans la spécification du domaine (fichier *.dspec*).

7.2.2 Consolidation des exigences

Dans cette étape, l'exigence rédigée en langage URM est consolidée[1] par les mots clés les plus utilisés lors de la spécification de propriétés. Pour la consolidation, nous nous basons sur un ensemble de patrons proposés par Dwyer [Dwyer *et al.* 1999], Cheng et Konrad [Konrad & Cheng 2005b]. Le tableau 7.1 regroupe l'ensemble de ces fragments. La consolidation consiste à remplacer les mots clés rencontrés dans les exigences par les mots clés du tableau 7.1.

Exemple:

La consolidation de l'exigence **AFS-SM-020** donne l'exigence consolidée suivante :

> ***In the context of*** *"the* SM *is in state Standby"*
> ***and*** *"the mission is not yet selected"*
> ***It is always the case that if*** *"a valid* EvtRequestLoginHmi *message from the* HMI*"* ***occurs***
> ***then*** *the* SM ***shall send*** *the following messages to the HMI* ***in this order*** *:*
> *(1)* EvtAckHmi ***with*** *result = TRUE, (2)* EvtCurrentUserHmi ***with*** Username = currentUsername*,*

Pour obtenir cette représentation de l'exigence **AFS-SM-020**, nous avons procédé comme suit:

1. D'abord, identifier le contexte dans lequel l'exigence doit être vérifiée. Dans ce cas, le contexte correspond à une condition sur l'état du système ainsi que la valeur d'une variable:

   State(SM) = Standby **and Val**(*IsMissionSelected*)=TRUE

2. Ensuite, identifier la propriété proprement dite et la reformuler en utilisant les mots clés correspondants aux fragments de patterns du tableau 7.1.

3. Enfin, choisir le type de séquencement des événements référencés dans les scénarios de l'exigence.

Ce processus est illustré à la figure 7.6.

[1] Nous avons choisi le mot "*consolider*" au lieu de compléter, car cette étape ne vise pas à ajouter plus d'informations dans les exigences issues de l'étape précédente

Table 7.1 : Grammaire du langage naturel pour les patrons de propriétés proposées par [Konrad & Cheng 2005a]

Start	1:property	::=	*scope "," specification "."*
Scope	2:scope	::=	"Globally" | "Before" R | "After" Q | "Between" Q and "After" R | "After" Q "until" R | "In the presence of " F | "In the absence of " F| "From when " F "never holds"
General	3:specification	::=	*qualitativeType* | *realtimeType*
Qualitative	4:qualitativeType	::=	*occurrenceCategory* | *orderCategory*
	5:occurrenceCategory	::=	*absencePattern* | *universalityPattern* | *existencePattern* | *boundedExistencePattern*
	6: absencePatten	::=	"it is never the case that " P " holds"
	7: universalityPattern	::=	"it is always the case that " P " holds"
	8: existencePattern	::=	P " eventually holds"
	9: boundedExistencePattern	::=	"transition to state in which " P " holds occurs at most twice"
	10: orderCategory	::=	"it is always the case that if " P " holds" (*precedencePattern* | *precedenceChainPattern1-2* *precedenceChainPattern2-1* | *responsePattern* *responseChainPattern1-2* | *responseChainPattern2-1* *constrainedChainPattern1-2*
	11: precedencePattern	::=	", then " S " previously held"
	12: precedenceChainPattern1-2	::=	"and is succeeded by " S ", the " T " previously held"
	13: precedenceChainPattern2-1	::=	", then " S " previously held and was preceded by " T
	14: responsePattern	::=	", then " S " eventually holds"
	15: responseChainPattern1-2	::=	", then " S " eventually holds and is succeeded by " T
	16: responseChainPattern2-1	::=	"and is succeeded by " S ", then " T " eventually holds after " S
	17: combinedChainPattern1-2	::=	", then " S " eventually holds and is succeeded by " T, " where " Z " does not hold between " S " and " T
Real-time	18: realtimeType	::=	"it is always the case that " (*durationCategory* | *periodicCategory* | *realtimeOrderCategory*)
	19: durationCategory	::=	"once " P " becomes satisfied, it holds for " (*minDurationPattern* | *maxDurationPattern*)
	20: minDurationPattern	::=	"at least " c " time unit(s)"
	21: maxDurationPattern	::=	"less than " c " time unit(s)"
	22: periodicCategory	::=	P " holds " *periodicOccurrencePattern*
	23: periodicOccurrencePattern	::=	"at least every " c " time unit(s)"
	24: realtimeOrderCategory	::=	"if " P " holds, then " S " holds " (*boundedResponsePattern* | *boundedInvariancePattern*)
	25: boundedResponsePattern	::=	"after at most " c " time unit(s)"
	26: boundedInvariancePattern	::=	"for at least " c " time unit(s)"

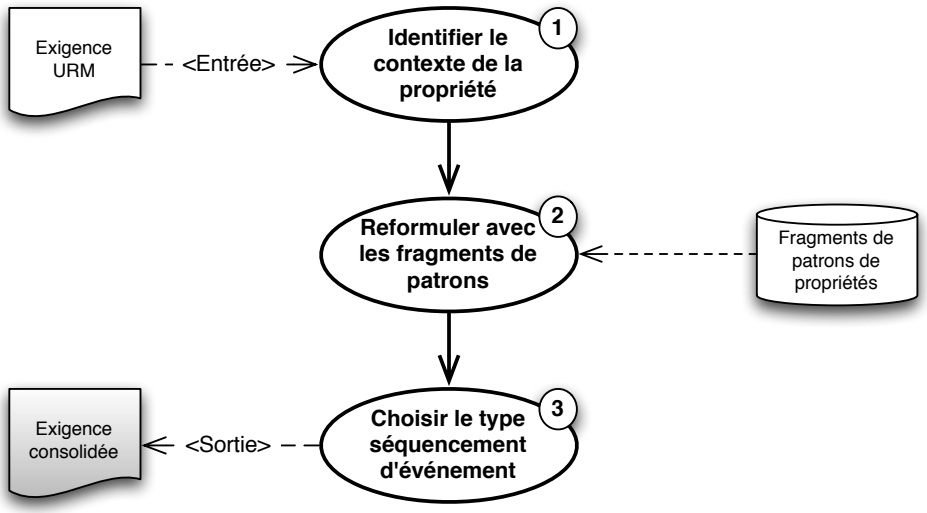

Figure 7.6 : Processus de consolidation des exigences URM

7

7.2.3 Analyse

La phase d'analyse correspond à une analyse structurelle des exigences issues de la phase de consolidation afin de déterminer le patron de propriété adéquat pour la génération de la propriété CDL. L'analyse se fait en comparant la structure de l'exigence utilisée avec l'ensemble des structures possibles dans le langage CDL.

En effet, une exigence est formalisée sous la forme d'une structure grammaticale correspondante aux structures proposées par le langage URM. Chaque phrase est formée de la concaténation de différents champs correspondants aux fragments de patrons. Le tableau 7.2 présente les structures de fragments de patrons du tableau 7.1 permettant la spécification des patrons de propriétés CDL ainsi que des exemples d'utilisation. Les numéros de la colonne "Structure" correspondent aux fragments de patrons de propriétés du tableau 7.1.

Table 7.2 : Mapping des structures d'exigences du tableau 7.1 sur les patrons de spécification de propriétés CDL

Patron	**Structures**	**Exemple**
Response	1, 2, 3, 10, 14/15/16	Globally, it is always the case that if P holds, then S holds.
Precedence	1, 2, 3, 10, 11/12/13	Globally, it is always the case that if P holds, then S previously held.
Abscence	1, 2, 3, 6	Before R, it is never the case that P holds.
Existance	1, 2, 3, 18, 19, 20/21	After Q, it is always the case that once P becomes satisfied, it holds for at least δ time unit(s)

La figure 7.7 présente l'algorithme d'analyse des exigences pour l'identification des patterns CDL adéquats à appliquer en vue de la génération des propriétés formalisées. L'algorithme commence par l'identification du scope de la propriété utilisé. Ce dernier correspond au champ de vérification de l'exigence comme nous l'avons illustré dans la figure 2.8 dans la section 2.3.4 sur les patrons proposés par Dwyer *et al.* dans [Dwyer *et al.* 1999]. Ensuite, une recherche de mots clés est lancée afin de déterminer la structure de l'exigence URM analysée. Si l'algorithme détecte l'emploi de mots clés temporels (tels que : "time unit, ms, duration. . . "), la condition temporelle est ajoutée. Sinon, l'exigence ne porte que sur l'occurrence ou l'ordre des événements. Une nouvelle recherche de mots clés est ensuite lancée afin de distinguer entre ces deux types d'exigences.

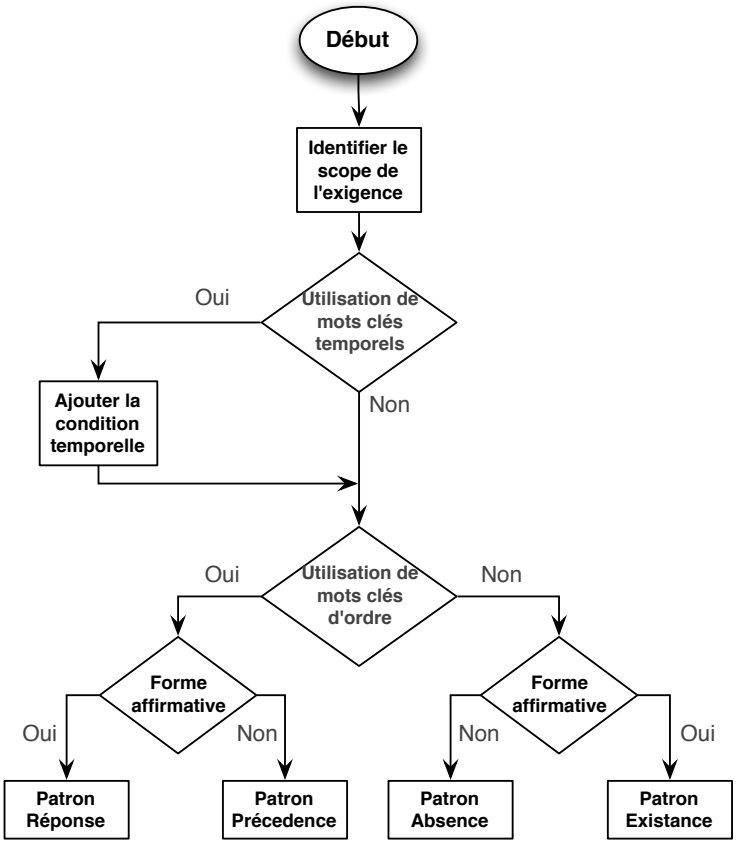

Figure 7.7 : Algorithme d'identification des patrons CDL dans les exigences URM

Exemple:

Pour illustrer cette étape, nous allons considérer l'exigence **AFS-SM-020** présentée précédemment.

1. L'algorithme commence par identifier le scope de l'exigence, dans ce cas, ce dernier est repéré par le mot clé : ***It is always the case***.

2. Une recherche des mots clés temporels ne produit aucun résultat (L'analyse de l'exigence ne détecte aucun mot clé tels que: time unit, ms...).

3. Le terme ***then*** indique qu'il s'agit d'un patron de type "*occurrence*".

4. Ensuite, Le mot clé ***then*** est suivie d'une phrase simple affirmative indiquant que l'exigence correspond à un "*patron réponse*".

Ainsi, selon le tableau 7.1, la structure de l'exigence **AFS-SM-020** est la suivante :

$$1 - 2 - 3 - 10 - 14$$

Ce qui correspond à un patron réponse (selon le tableau 7.2).

7.2.4 Rectification

Dans le cas où aucune structure de patrons CDL n'est identifiée dans l'exigence proposée par l'expert métier, ce dernier est invité à revoir la structure de l'exigence afin de corriger les éventuelles incohérences. Une incohérence dans les exigences spécifiées dans URM peut résulter du fait que le *scope* n'as pas été spécifié.

7.2.5 Instanciation

La dernière étape du processus de formalisation des exigences proposée dans ce mémoire consiste à traduire l'exigences URM en propriété CDL en se basant sur le patron de propriété identifié au cours de la phase d'analyse. Ainsi, chacun des champs de l'exigence est traduit par transformation de modèle vers la syntaxe CDL.

En effet, la syntaxe des propriétés CDL est illustrée dans la figure 7.8. Celle-ci est présentée avec les conventions suivantes:

`property_label`	::=	nom de la propriété
`optional_expression`	::=	chaine de caractères
`event_name`	::=	identifiant d'un événement
`delayExpression`	::=	contrainte temporelle

Aussi, au cours de cette phase, nous procédons à l'éclatement de l'ensemble des événements qui sont échangés avec plusieurs valeurs d'arguments ainsi que les composants qui sont instanciables plusieurs fois lors des interactions. En effet, dans l'exigence **AFS-SM-020**, on trouve par exemple :

> ...the *SM* shall send the following messages to the HMI in ...

Dans la spécification du domaine, l'acteur HMI possède trois instance "HMI (1..3)". Dans de cas, lors de l'instanciation, les messages seront envoyés aux trois instances de l'acteur HMI. Ainsi, l'instanciation de l'exigence **AFS-SM-020** donnera la propriétés CDL de la figure 7.9.

Patron *Response*

```
Property property_label <optional_expression>
{
  Occurrence_Expression
  (immediately | eventually) leads-to <delayExpression>
  Occurrence_Expression
  <event_name> (may never | must) occur
                  [(,<event_name> (may never | must) occur)*]
  (one of <event_name>[(,<event_name>)*] (cannot | may)
   occur before the first one of <event_name>[(,<event_name>)*])*
  repeatability : true | false
}
```

Patron *Absence*

```
Property property_label <optional_expression>
{
  Occurrence_Expression
  occurs never <delayExpression>
  repeatability : true | false
}
```

Patron *Precedence*

```
Property property_label <optional_expression>
{
  Occurrence_Expression
  (immediately | eventually) precedes <delayExpression>
  Occurrence_Expression
  <event_name> (may never | must) occur
                  [(,<event_name> (may never | must) occur)*]
  (one of <event_name>[(,<event_name>)*] (cannot | may)
   occur before the first one of <event_name>[(,<event_name>)*])*
  repeatability : true | false
}
```

Patron *Existance*

```
Property property_label <optional_expression>
{
  Occurrence_Expression
  occurs  < delayExpression >
  repeatability : true | false
}
```

```
Occurrence_Expression ::= {
                          AN | ALL ordered | ALL combined
                          {
                            (exactly one | one or more) occurrence[s] of  <event_name>)*
                          | (if (<condition_expression>)
                          {
                          (exactly one | one or more) occurrence[s] of <event_name> )* })*
                          }
```

Figure 7.8 : Syntaxe concrète du langage de spécification de propriétés CDL

Code CDL

```
Property AFS-SM-020
{
        IF (scope A)
        ALL combined{
                An exactly one occurrence of EvtRequestLoginHmi_1
                Or
                An exactly one occurrence of EvtRequestLoginHmi_2
                Or
                An exactly one occurrence of EvtRequestLoginHmi_3
             }

                immediately leads-to

        ALL combined{
                An exactly one occurrence of EvtAckHmi_1(TRUE)
                An exactly one occurrence of EvtCurrentUserHmi_1(currentUsername)
                And
                An exactly one occurrence of EvtAckHmi_2(TRUE)
                An exactly one occurrence of EvtCurrentUserHmi_2(currentUsername)
                And
                An exactly one occurrence of EvtAckHmi_3(TRUE)
                An exactly one occurrence of EvtCurrentUserHmi_3(currentUsername)
        }
        ENDIF

        EvtAckHmi_1 may never occur before the first one of EvtCurrentUserHmi_1
        And
        EvtAckHmi_2 may never occur before the first one of EvtCurrentUserHmi_2
        And
        EvtAckHmi_3 may never occur before the first one of EvtCurrentUserHmi_3
}

Scope A
{
        state (SM) = Standby  and
        eval (MissionSelected) = FALSE
}
```

Figure 7.9 : Propriétés CDL correspondant à l'exigence **AFS-SM-020**

7.3 Discussion et synthèse

Nous avons proposé dans ce chapitre notre approche pour la génération de code CDL à partir des langages orientés utilisateurs proposés dans la deuxième partie de ce mémoire. Nous avons proposé de spécifier les contextes à l'aide de cas d'utilisation étendus pour spécifier les interactions entre le modèle à valider et les différents acteurs de l'environnement.

En effet, suite à la capture de ces interactions sous forme de scénarios, la difficulté consiste à synthétiser le comportement des différents acteurs afin de produire le modèle CDL du contexte. Dans ce chapitre, nous avons présenté les algorithmes permettant une telle synthèse. D'abord, nous avons transformé les scénarios capturés par les cas d'utilisation étendus en des diagrammes d'activités. Chacun de ces diagrammes d'activités représente le comportement global du cas d'utilisation étendu. Les actions relatives à chaque acteur ont été placées dans une partition d'activité qui lui est propre (*uml::ActivityPartition*). Ensuite, la deuxième transformation se base sur ces partitions d'activités pour isoler le comportement de chaque acteur dans une activité toute préservant le comportement global du cas d'utilisation source. Cette préservation est garantie par l'ajout de noeuds spéciaux (*uml::AcceptEventAction* et *uml::SendSignalAction*).

Pour compléter le modèle CDL généré, nous avons aussi proposé de spécifier les exigences par du langage naturel contrôlé (langage URM). À partir de cette spécification, nous avons proposé une chaine de transformations de modèle pour la génération des propriétés CDL en cinq étapes. Cet algorithme de formalisation est basé sur l'utilisation des patrons de spécification de propriétés. L'utilisateur est assisté par un référentiel de fragments des patrons de spécification. La réécriture des exigences consiste donc à choisir les fragments adéquats et de les composer pour obtenir une exigence URM *consolidée*. Les exigences sont ainsi le résultat de la concaténation de ces fragments. L'algorithme proposé se base sur l'identification de ces fragments afin de déterminer quel est le patron de spécification de propriété qu'il faut utiliser afin de formaliser l'exigence considérée. La dernière étape du processus consiste à instancier la propriété CDL correspondante au patron détecté.

Toutefois, pour garantir une bonne intégration des différentes étapes proposées dans ce chapitre aux processus de développement, nous avons besoin de définir un cadre méthodologique. Par conséquent, nous proposons dans le chapitre suivant une méthodologie d'utilisation des concepts proposés dans ce mémoire au sein d'un processus industriel.

8 Méthodologie de vérification et intégration aux processus de développement

Nous avons présenté, dans le chapitre 2, un état de l'art sur les processus de développement des systèmes embarqués temps-réel. Suite à l'étude de ces processus nous avons fait le constat que peu d'entre eux précisent la manière avec laquelle les activités de vérification formelles sont intégrées aux activités de développement. De ce fait, dans ce chapitre, nous présentons notre méthodologie de vérification permettant l'intégration des activités proposées dans les chapitres précédents aux processus de développement industriel. D'abord, nous présentons les éléments permettant de définir une méthodologie dans la section 8.1, puis nous présentons la méthodologie proprement dite dans la section 8.2.

8.1 Définition et formalisation d'une méthodologie

Le mot méthodologie utilisé dans ce chapitre correspond à la description d'une succession d'activités de travail qui gouvernent la production de modèles formels assimilables par les outils existants de vérification [Colombo *et al.* 2007]. Pour ce qui est du processus de développement, celui-ci correspond aux activités mises en oeuvre au sein d'une équipe de développement pour la production de biens livrables. Ces biens livrables correspondent aux modèles de conception que l'on désire valider formellement afin de s'assurer du respect des exigences exprimées au début du cycle de développement.

8.1.1 Vue d'ensemble de la méthodologie

La méthodologie que nous proposons dans cette thèse intègre les activités liées à la vérification formelle des modèles de conception. En ce sens, elle raffine les méthodologies de conception existantes pour y intégrer les préoccupations de formalisation et de génération de code formel. Le but étant de pouvoir utiliser les outils de vérification formelle existants tout au long du processus de développement. L'approche que nous proposons dans cette thèse a été définie de manière générique pour s'appliquer aux principaux processus de développement utiliser dans le contexte du développement de système logiciels embarqués.

En effet, nous avons présenté dans la section 2.4 deux des processus de développement les plus répandus dans ce contexte : le processus unifié UP [Booch *et al.* 1999] et le processus de développement en "V". À partir de ces deux processus, nous avons dégagé les activités de développement communes afin d'y intégrer les activités proposées dans

notre approche. Ces activités sont : (1) le traitement des exigences (2) Spécification des cas d'utilisation (3) l'analyse et (4) la validation.

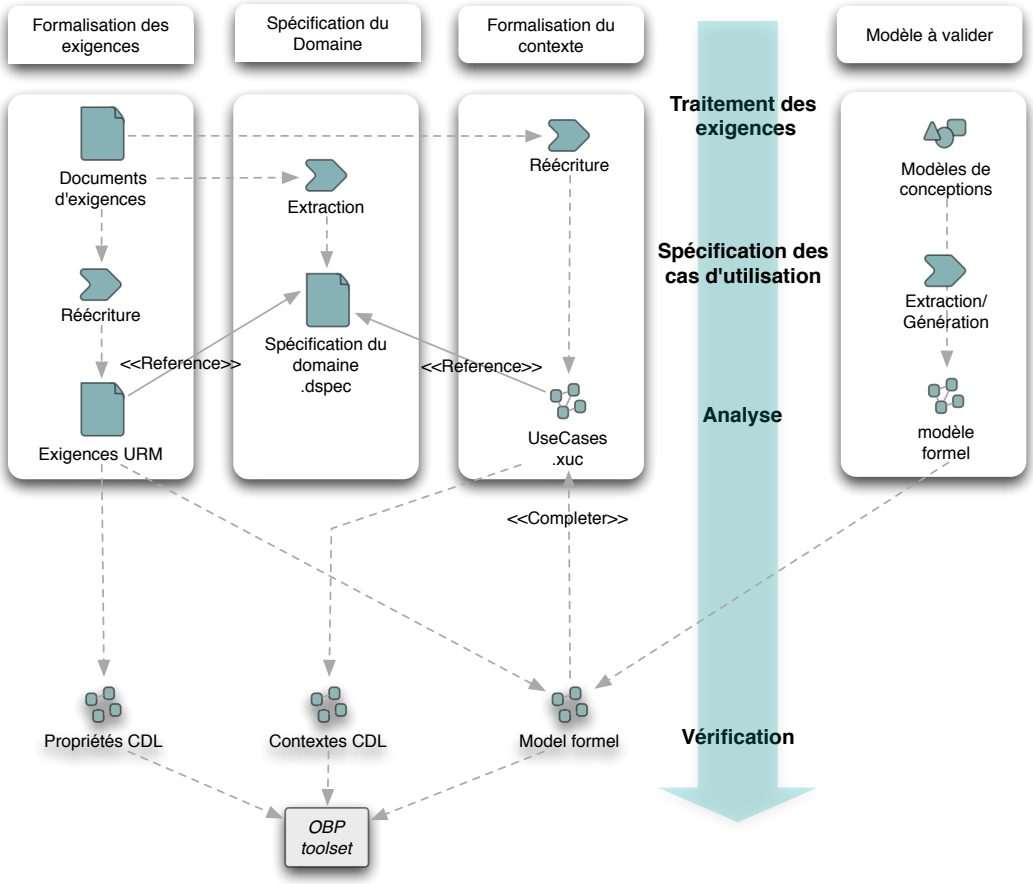

Figure 8.1 : Vue générale de la méthodologie proposée

- **Le traitement des exigences:** cette activité concerne la formalisation des données d'entrées à l'aide des langages proposés dans la deuxième partie de ce mémoire,
- **La spécification des cas d'utilisation:** permet de formaliser les scénarios d'interaction entre le composant du système à vérifier et son contexte,
- **L'analyse:** consiste en la génération de modèles CDL à partir des modèles utilisateurs produits par la phase de traitement des exigences,
- **La vérification:** concerne l'exploitation des outils de vérification formelle pour obtenir le résultat.

Ainsi, la figure 8.1 présente la méthodologie proposée et l'organisation des activités de vérification proposée selon les activités des processus de développement. Le point

d'entrée de celle-ci est un ensemble d'exigences non formalisées ainsi que le modèle de conception préliminaire du système étudié. Ces modèles regroupent les diagrammes de cas d'utilisation et de séquences qu'on trouve dans les documents de spécification pour illustrer et clarifier les exigences sur les fonctionnalités attendues du système. Pour que le modèle à vérifier soit exploité par la méthodologie proposée, celui-ci doit être décrit par un langage formel exploitable par l'outil de vérification formel utilisé. Dans le cas de notre travail, le modèle à validé est présenté sous la forme d'un programme FIACRE pour être vérifier par l'outil TINA. L'aspect formalisation du modèle du système à vérifier n'entre pas dans le périmètre de nos travaux, menés dans le cadre de cette thèse.

Dans la suite de ce chapitre, nous allons détailler chacune des quatre activités méthodologiques. Mais avant de procéder, nous allons présenter la notation de description de méthodologie utilisée dans la section qui suit.

8.1.2 Notation utilisée pour la description de la méthodologie

La notation utilisée dans ce chapitre pour décrire un processus de développement méthodologique étend les diagrammes d'activités UML2 [OMG 2007]. Nous reprenons la syntaxe concrète utilisée dans [Fontan 2008] pour décrire notre méthodologie. En effet, dans ce dernier, les étapes méthodologiques sont décrites par des activités qui peuvent être raffinées en sous activités. Un diagramme d'activité commence obligatoirement par un disque noir représentant le début de la première activité. Chaque étape est reliée par une flèche de transition montrant les relations de précédence entre les activités. Une activité peut être décomposée en sous activités. La figure 8.2 montre un exemple de processus de développement méthodologique.

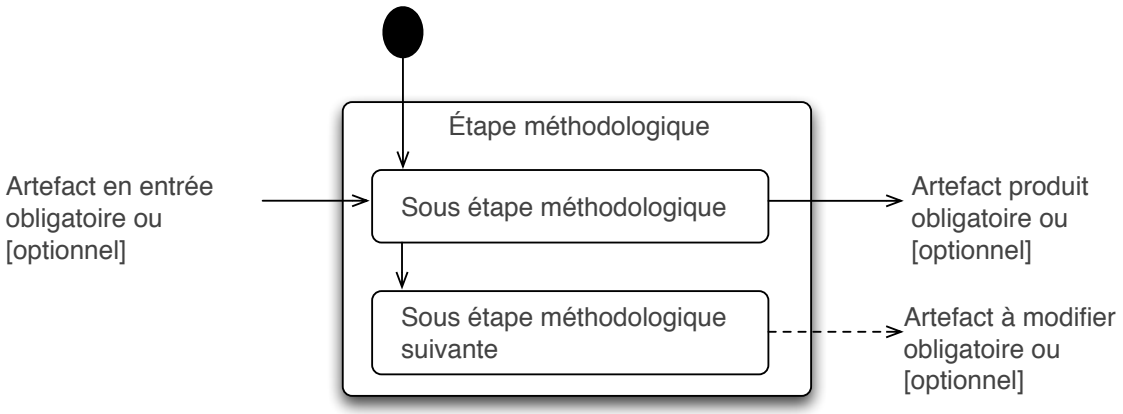

Figure 8.2 : Langage de description de la méthodologie

Lors des étapes de la méthodologie, des artefacts peuvent être produits, consommés ou modifiés par une ou plusieurs étapes. La production (création/modification) et la consommation (consultation) sont respectivement représentées par des flèches pleines (par opposition aux flèches de transition entre activités) sortantes et entrantes de l'activité sur les côtés. Les artefacts modifiés lors d'une activité méthodologique sont représentés par

une flèche sortante marquée en trait discontinu. Un artefact sortant est disponible pour toutes les autres activités, contrairement à la production d'un artefact qui n'est autorisé qu'à l'intérieur des activités qui possèdent une flèche sortante.
Le diagramme d'activités représentant la méthodologie est ainsi composé d'étapes méthodologiques, de référence vers les artefacts utilisés par ces activités et de flèches reliant ces derniers. Ainsi, la méthodologie proposée définit un ensemble d'activités méthodologiques et associe à chacune de ces étapes la livraison d'artefacts. La notion d'artefact ici fait référence à des modèles conformes à des métamodèles.

8.2 Application à la vérification formelle des exigences et l'exploitation des contextes

La méthodologie proposée est dédiée à la vérification formelle de modèles et ne couvre donc pas les phases de conception et d'implémentations. Le but de celle-ci est d'intégrer la production de modèles formels nécessaires à l'utilisation des outils existants de *model checking*. Comme nous l'avons précisé précédemment, nous nous plaçons dans le cadre de la vérification par exploitation de contextes, et notamment en utilisant des modèles CDL ainsi que la chaine d'outils OBP développés dans le laboratoire.

8.2.1 Traitement des exigences

Dans notre approche, la première activité est l'étude des documents d'exigences par l'expert métier afin de les réécrire sous la forme d'exigences URM. Nous supposons que les exigences définies dans ces documents, fournis par le client, sont des exigences de bas niveau (par opposition aux objectifs de haut niveau). En effet, chaque exigence devra porter sur une fonctionnalité simple du système étudié afin de vérifier des contraintes telles que le séquencement d'événements ou des contraintes temporelles sur les messages échangés. Comme nous l'avons préciser au par avant, la décomposition des exigences étant une problématique à part entière, elle n'entre pas dans le cadre de cette thèse.

La figure 8.3 illustre l'activité méthodologique de la phase de traitement des exigences.

8.2.2 Spécification des cas d'utilisation

Cette activité concerne l'exploitation des documents de spécification afin d'identifier les différents cas d'utilisation du système. Ces cas d'utilisation sont ensuite détaillés afin de spécifier les scénarios d'interaction entre le système et les entités de son environnement (acteurs). À ce stade, les scénarios d'interactions peuvent ne référencer que les séquences d'échange nominal entre le système et le contexte. L'information permettant de spécifier de tels scénarios est généralement définie dans les cahiers de charge. Par la suite, ces scénarios peuvent être raffinés afin d'y intégrer les différentes exceptions potentielles qui peuvent altérer le déroulement du scénario nominal. Selon les exceptions identifiées, des *handlers* peuvent être définis pour enrichir la spécification du scénario principal.

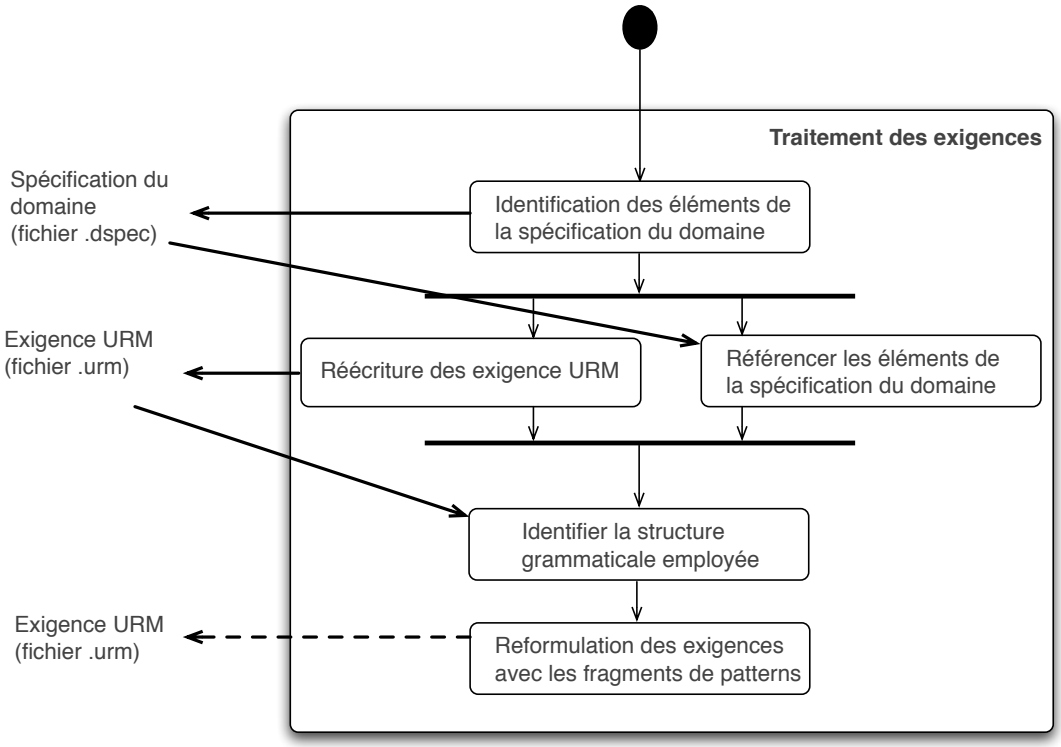

Figure 8.3 : Activité de traitement des exigences

La figure 8.4 détaille cette phase de la méthodologie.

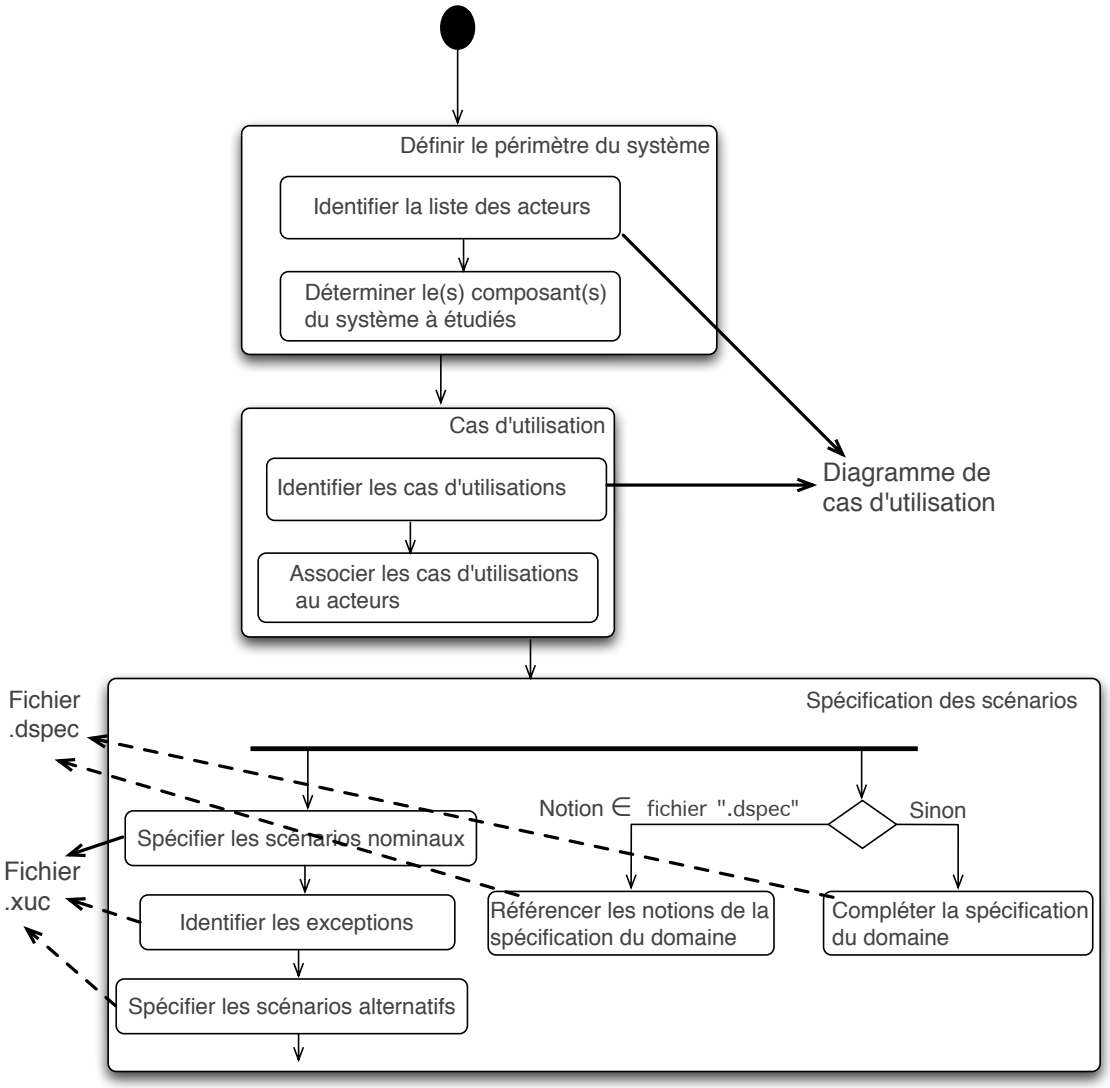

Figure 8.4 : Activité de spécification des cas d'utilisation

8.2.3 Analyse des modèles formels

Cette phase est implémentée sous forme de transformations de modèles permettant la génération de modèles CDL. Les propriétés CDL sont générées à partir des exigences URM issues de la phase de traitement des exigences selon le processus détaillé dans la section 7.2. En ce qui concerne les modèles comportementaux des acteurs, ceux-ci sont obtenus à partir des modèles XUC produits lors de l'activité de spécification des cas d'utilisation. La figure 7.1 présentée au début du chapitre 7 illustre cette activité.

8.2.4 Vérification

La vérification des propriétés formalisées à partir des exigences sur le modèle du système étudié en exploitant les contextes se fait à l'aide de la chaine d'outils OBP [Dhaussy *et al.* 2009]. Au cours de cette phase, les modèles CDL sont traduits, dans l'outil OBP, en codes assimilables par un vérificateur. Dans nos expérimentations, OBP génère des programmes Fiacre [Berthomieu *et al.* 2008] qui sont exploités soit par l'explorateur OBP (*OBP Explorer*) soit par l'outil TINA [Berthomieu *et al.* 2004]. Ensuit, OBP récupère les résultats retournés par ces composants afin d'évaluer la correction de la propriété. La figure 8.5 présente une vue d'ensemble de cette chaine d'outils.

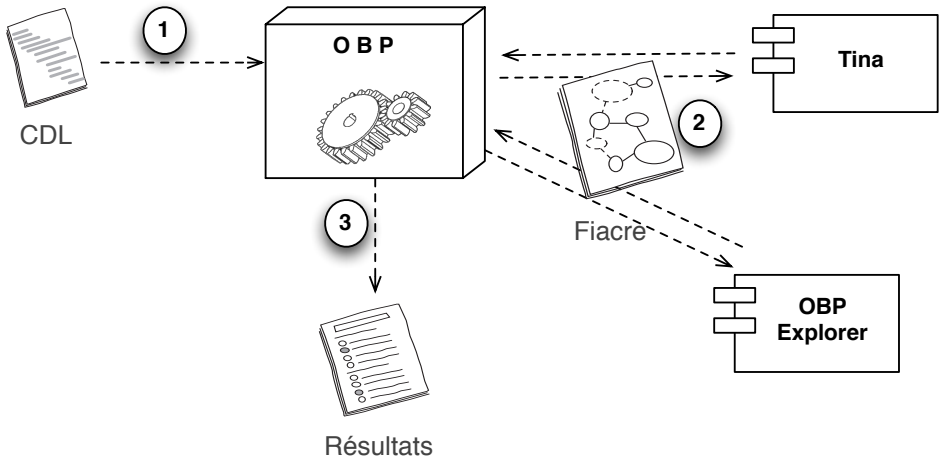

Figure 8.5 : L'outil OBP

8.3 Discussion et synthèse

Nous avons présenté dans ce chapitre une méthodologie permettant l'intégration des activités liées à la vérification formelle par exploitation de contextes dans un processus de développement logiciel. L'originalité de notre méthodologie consiste à proposer aux experts métier des formalismes adaptés à la formalisation des contextes et des exigences dès les premières étapes du processus de développement. Ces formalismes, que nous avons présentés dans la deuxième partie de ce mémoire, réduisent le gap entre les modèles manipulés lors des premières étapes de développement et ceux nécessaires à la vérification formelle. Ainsi, au cours de cette méthodologie, les modèles de contextes et des exigences peuvent être raffinés de façon incrémentale à partir des cas d'utilisation et des exigences textuelles.

Aussi, la chaîne d'outils OBP constitue une interface commune pour la génération de code formel vers le *model checker* ciblé. En effet, OBP intègre les transformations de modèles nécessaires pour générer, à partir d'un même modèle CDL soit du code FIACRE pour utiliser l'outil TINA, soit du code IF [Bozga *et al.* 1999] afin d'exploiter le simulateur IFx. Les retours de preuve d'OBP ne sont pas pris en charge dans notre approche. Toutefois,

ce travail est en cours au sein d'équipe IDM de l'ENSTA-Bretagne afin d'intégrer la prise en charge de ces retours et de les présenter aux utilisateurs directement dans les modèles de haut niveau.

Nous évaluons la pertinence et l'applicabilité de notre méthodologie pour valider le modèle d'un composant logiciel industriel réel. Le chapitre suivant présente cette évaluation.

EXPÉRIMENTATIONS ET CONCLUSION

TELECOM Bretagne

9 Cas d'Étude Industriel: Validation du composant AFS_SM

Nous appliquons l'approche de vérification proposée dans ce mémoire sur le cas d'étude présenté dans le chapitre 3. Ce cas d'étude a été utilisé le long de ce mémoire afin d'illustrer les concepts et les algorithmes proposés. Dans ce chapitre, nous allons détailler l'ensemble des étapes de la méthodologie proposée dans le chapitre précédent pour vérifier les exigences du système en considérant ses cas d'utilisation. Le but de cette expérimentation est de montrer l'applicabilité de notre approche et la contribution apportée pour améliorer l'intégration des activités de vérification formelle aux processus de développement utilisés chez notre partenaire industriel. Ce chapitre est constitué de trois parties : La première partie est une description des exigences et des cas d'utilisation fournis en entrée par notre partenaire industriel. La deuxième partie est une application de la méthodologie de vérification présentée dans ce manuscrit sur le cas d'étude. Dans la troisième partie, nous étudions l'apport de notre approche par rapport à une application classique des techniques de vérification formelle.

9.1 Formalisation des cas d'utilisation

Cette section commence par la présentation les artefacts fournis par notre partenaire industriel pour la validation du composant SM du système AFS. Puis, nous présentons les différents cas d'utilisation et le processus de leur formalisation à l'aide du langage XUC. Ce processus de formalisation débouche sur la génération de la partie contexte des programmes CDL. Celle-ci est utilisée, par la suite, pour la vérification des propriétés issues des exigences. Cette dernière étape est détailler dans la section 9.2.

Une description des différents composants du système a été faite dans la section 3.1. Ce chapitre, se focalise sur les cas d'utilisations formalisés à travers le langage XUC ainsi que les exigences traitées.

En effet, les fonctions techniques du SM ont pour objectif de fournir les capacités opérationnelles pour répondre aux exigences des utilisateurs. Parmi ces fonctions techniques ont trouve:

- Lancer une mission
- Arrêter une mission
- Arrêter le système

- Gérer les opérations conduites par les opérateurs: *login*, *logout*, sélectionner une mission, sélectionner un rôle. . . .

Le diagramme des cas d'utilisation du système, présenté dans la spécification technique est présentée dans la figure 9.1.

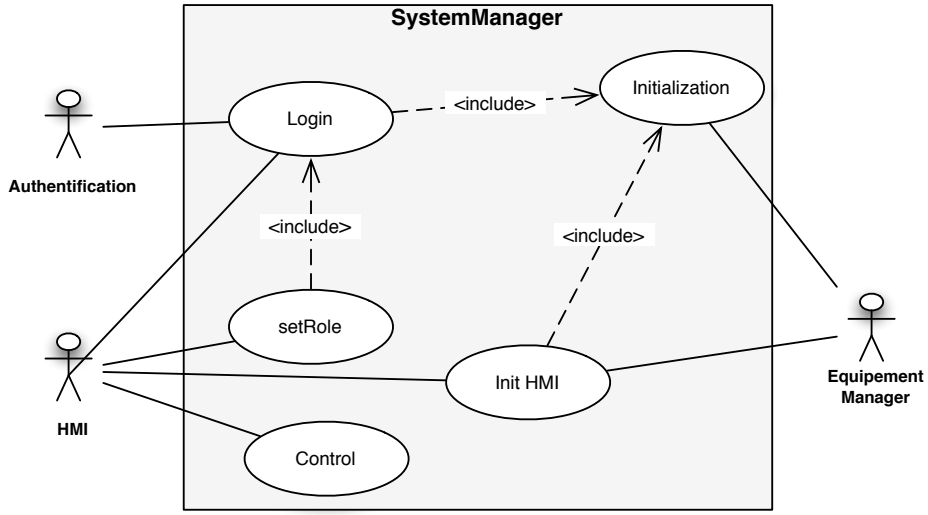

Figure 9.1 : Cas d'utilisation du composant *SM*

Le système AFS a été modélisé par les ingénieurs de notre partenaire industriel en utilisant l'outil *Rhapsody*. Par la suite, une implémentation Java a été générée depuis les modèles *Rhapsody*. La description des scénarios d'interactions entre le système et son contexte pour la réalisation des différentes fonctions techniques est présentée dans la spécification technique sous la forme de diagramme de séquence *Rhapsody*. Un exemple de ces diagrammes est présenté à la figure 9.2.

Dans ce diagramme, on trouve la description des scénarios d'interaction liés à trois cas d'utilisation:

- **Initialisation:** l'interaction correspondante à ce scénario est référencée en utilisant le mot clé *"Ref"* proposé par le standard UML2,
- **Login:** permettant l'authentification des utilisateurs par identifiant et mot de passe. Ce scénario débouche sur le changement d'état du composant *SM* qui passe de l'état *Idle* à l'état *Logged*,
- **Sélection de missions:** (*setRole*) permettant d'attribuer une mission (donc un rôle) à l'utilisateur authentifié.

Donc, un premier travail consiste à identifier les interactions propres à chaque cas d'utilisation. Ainsi, la formalisation de ce scénario d'interaction est réalisée en formalisant les trois cas d'utilisations référencés en utilisant le langage XUC.

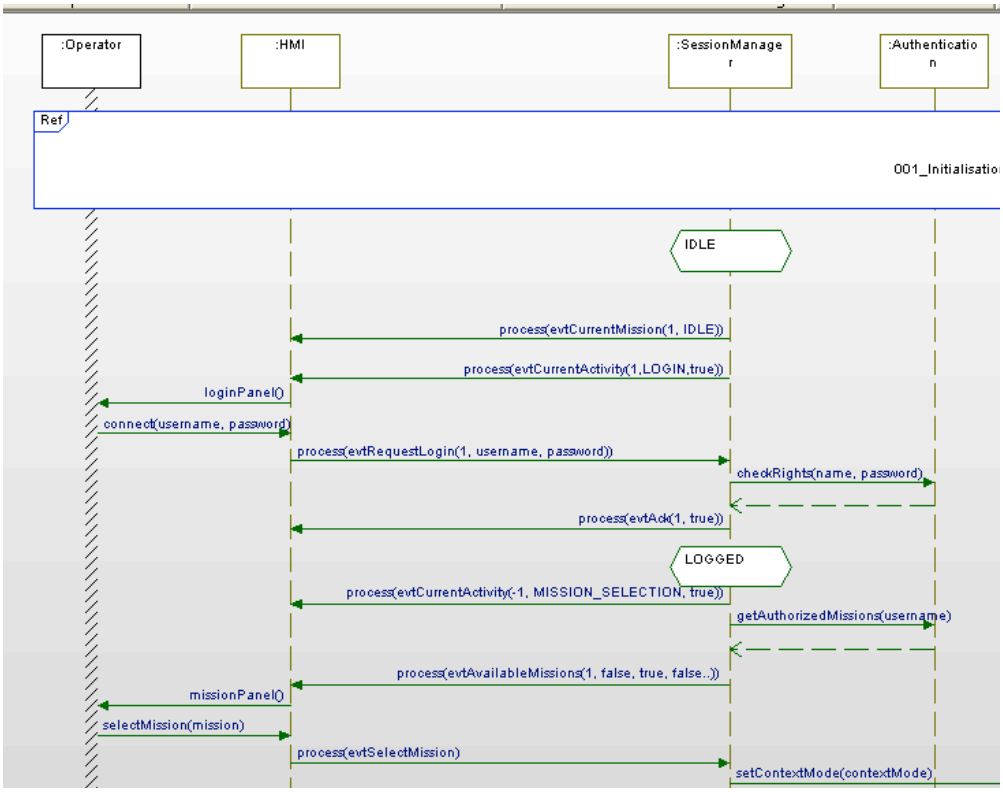

Figure 9.2 : Exemple de scénario fournis dans les documents de spécification

En effet, le composant *SM* permet aux opérateurs de lancer une mission, d'arrêter une mission ou d'éteindre le système. Il est aussi responsable de la gestion des profils des utilisateurs (contrôle d'accès, attribution des rôles). Les opérateurs peuvent interagir avec ce composant en envoyant des requêtes à travers des consoles appelées *HMI*. Les consoles envoient des requêtes au composant *SM* au moyen de communications asynchrones. Ce dernier répond par l'affirmative ou par la négative et renseigne au besoin les *HMIs* de l'évolution du système. Dans certains cas d'utilisation, le *SM* doit communiquer avec d'autres acteurs, comme l'authentification (Authentification) ou le gestionnaire d'équipements (*Equipement Manager*).

À titre d'exemple, les requêtes envoyées au SM via les HMIs sont :

- Se connecter
- Se déconnecter
- Sélectionner une mission (avec un contexte guerre/paix et un mode combat/entrainement)
- Sélectionner un ou plusieurs rôles
- Stopper une mission en cours
- Passer de la préparation à l'opération et inversement

- Eteindre le système

Quant aux informations envoyées par le *SM*, on trouve:

- Confirmation (« acknowledgement »)
- Activité courante
- Mission courante
- Les requêtes accessibles

L'ensemble de ces interactions ainsi que les arguments échangés ont été répertoriés au sein d'une spécification du domaine propre au composant SM du système AFS. Cette spécification du domaine servira aussi lors de la phase de réécriture des exigences en langage URM. Au cours de nos expérimentations, cette spécification du domaine a été construite manuellement à partir de la documentation technique du système ainsi qu'à partir des modèles UML (diagrammes de classes et des machines à états). Une vue partielle de cette spécification du domaine construite pour ce cas d'étude est présentée à la figure 9.3.

SystemManager.dspec

```
DomaineSpecification
{
    Actors{
        EquipmentMgr:: 'Gestionnaire d"equippements'
        Authentification:: 'Gestionnaire d"authentification'
        HMI:: 'Console'
    }

    SystemElements{
        SM:'System Manager'
    }

    Notions{
        //Consoles <-> SM
        Message login :'demande de login des console vers le SM'
        Message ackLoginOK :'acquittement du login reussit'
        Message ackLoginKO :'acquittement du login échoué'
        Message currentUser : 'spécifier l"utilisateur courant'
        Message logout :'demande de logout des console vers le SM'
        Message ackLogout :'acquittement du logout'
        Message EvtRequestMissionPhaseHmi : 'Demande de mission'
        Message EvtAckHmi : 'Acquitement des HMI'
        Message EvtMissionPhaseHmi : 'Reponse de mission'
        Message currentUserID : 'Spécifier l"identifiant de l"utilisateur courrant'
        Message currentUserMission : 'Spécifier l"identifiant de la mission courran
        //Authentification <-> SM
        Message check : 'vérifier les identifiants'
        Message ackCheckOK : 'acquittement reussit de la vérification des identific
        Message ackCheckKO : 'acquittement échoué de la vérification des identifica

        State Active_Preparation : 'Active preparation state'
        State Active_Operational : 'Active operational state'
        State Operational : 'Operational state'
```

Figure 9.3 : Spécification du domaine spécifique au composant SM

La construction des modèles de contexte a nécessité l'étudie les artefacts décrivant les cas d'utilisation du composant SM. Nous disposions au départ de l'étude :

- De diagrammes de Use Cases UML Rhapsody
- De diagrammes de séquence UML Rhapsody

- De machines à états UML Rhapsody
- De séquences JUnit
- Du code Java
- D'exigences textuelles.

Toutefois, nous avons aussi constaté que les diagrammes UML n'étaient pas à jour par rapport au code java fourni. En effet, les codes java générés ont été manuellement modifiés. Nous nous sommes donc basés sur ce code java et notamment sur les séquences JUnit afin d'identifier les interactions entre le composant SM et son contexte.

Ainsi, sur les cinq cas d'utilisation présentés dans la figure 9.1, nous avons spécifié les scénarios nominaux ainsi que les différentes exceptions que peut rencontrer le composant SM au cours de son cycle de vie. La figure 9.4 présente les trois XUC correspondant au scénario de la figure 9.2. Nous avons utilisé la relation *"include"* entre les XUC (présentée à la section 5.3).

```
XUC login

body {
    include Initialization
    step s1: HMI sends evtRequestLogin to SM
    step s2: SM sends checkRights to Authentification
    step s3: SM receives ackEvtRequestLogin from Authentification
    step s4: SM sends evtAvaliableMissions to HMI
}

exception badIdentifier{
    description 'The user enter wrong identifier/password three times'
    relatedStep s2
    relatedHandler badIdentifierHandler
    next s1
    outcome failure
}

handler badIdentifierHandler{
    step h1: SM sends ErrorFailedMessage to HMI
    step h2: SM sends LockAccount to Authentification
}
```

```
XUC setRole

body {
    include login
    step s1: SM sends evtCurrentActivity to HMI
    step s2: SM receives ackevtCurrentActivity from HMI
    step s3: SM sends evtAvalibleMissions to HMI
    step s4: HMI sends evtSelectMission to SM
    step s5: SM sends setContextMode to Authentification
}
```

```
XUC Initialization

body{
    step s1: HMI sends login to SM
    step s2: SM sends check to Authentification
    step s3: Authentification sends ackCheckOK to SM
    step s4: SM sends ackLoginOK to HMI
    step s5: SM sends currentUserID to HMI
    step s6: SM sends currentUserMission to HMI
}

exception MessageLost{
    description 'In that case that the response to a message
                 is not received within d_max duration'
    relatedStep s3,s4
    relatedHandler MessageLostHandler
    next s2
}

handler MessageLostHandler{
    step h1: SM sends check to Authentification
}
```

Figure 9.4 : Exemple de modèles XUC construit pour le SM

Suite à cette spécification, nous avons appliqué les algorithmes présentés dans les chapitres précédents afin de générer les modèles CDL décrivant les comportements des acteurs *HMI*, *Authentification* et *Equipment Manager*.

L'ensemble des cas d'utilisation de la figure 9.1 ont été formalisés en utilisant le langage XUC pour décrire les interactions entre le SM et son environnement. Au cours de cette formalisation, les différentes exceptions détaillées dans la documentation de

spécification ont été intégrées aux scénarios concernés. De ce fait, pour les cinq cas d'utilisation présentés dans la figure, 12 modèles CDL ont été générés. Ces modèles correspondent aux comportements des trois acteurs participants dans les différents cas d'utilisation.

En effet, pour la génération de la partie contexte, les modèles XUC ont été traduits en modèles UML à travers une transformation en deux temps; (1) traduction des scénarios XUC en diagrammes d'activités (un seul diagramme d'activité généré par XUC), puis (2) séparation des comportements des acteurs participant au cas d'utilisation dans des diagrammes d'activités séparés. Ces deux transformations ont été implémentées en langage de transformation QVT [OMG 2008a] comme nous l'avons illustrer dans la figure 7.1.

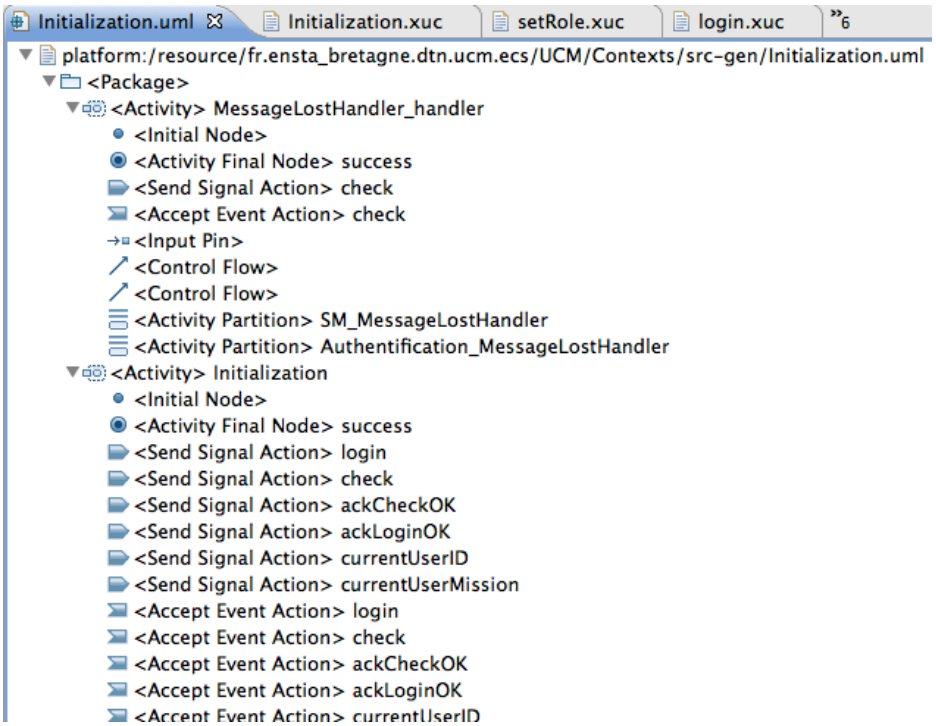

Figure 9.5 : Diagramme d'activité du XUC *Initialization*

La figure 9.5, présente les diagrammes d'activité générés pour le XUC *Initialization* de la figure 9.4. En effet, on avons généré un diagramme d'activité par scénario: un pour le scénario principale et un autre pour le scénarios décrit par le *handler* (comme nous l'avons expliquer à la section 7.1.2). Ces deux diagrammes représentent le comportement globale du XUC correspondant.

La deuxième transformation nous génère un diagramme d'activité pour chaque acteur participant au cas d'utilisation selon l'algorithme présenté à la section 7.1.2.2. La figure 9.6 présente le modèle UML décrivant les activités générés.

La dernière étape consiste à traduire les diagrammes d'activités représentant les comportements des acteurs en programmes CDL. Cette transformation, modèle vers texte, a

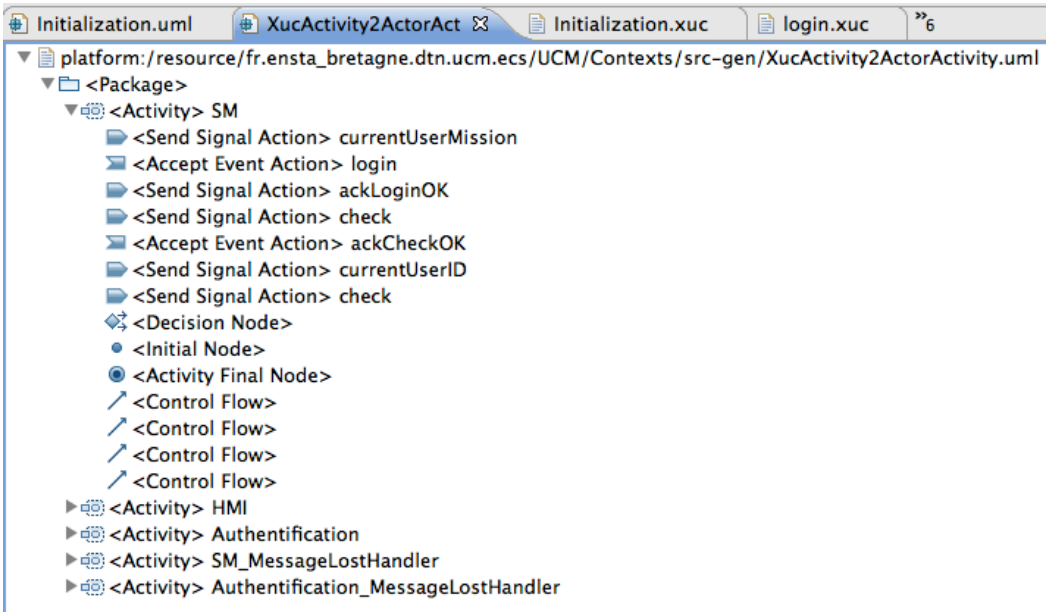

Figure 9.6 : Modèle UML généré par la deuxième transformation

été implémenté à l'aide du langage de génération de code *Xpand*[1].

9.2 FORMALISATION DES EXIGENCES

Les exigences fournies dans les cahiers des charges sont au nombre de 70. La première colonne du tableau 9.1 présente les dix exigences traitées lors de notre expérimentation. En effet, plusieurs exigences rencontrées dans le cahier des charges se ressemblent d'un point de vue structurel, et donc leur formalisation suit le même schéma. Nous avons choisi ce sous-ensemble d'exigence de façon à ce qu'il soit représentatif de l'ensemble des exigences exprimées.

Dans cette phase, nous avons réécrit les exigences présentées dans le tableau 9.1 afin de préparer la génération automatique des propriétés CDL. Puis, nous avons appliqué le processus de spécification des exigences présenté dans la figure 7.4 à la page 103.

Cette activité démarre par l'identification des éléments de la spécification du domaine. Celle-ci fournira en sortie un ou plusieurs fichiers DSpec listant les différentes notions du domaine. Lors de cette phase, nous avons utilisé l'éditeur des exigences URM, développé dans le cadre de cette thèse, afin de réécrire les exigences présentées dans le cahier des charges. Comme le montre la figure 8.3, une spécification du domaine est utilisée au cours de cette phase. Cette étape aboutit à la construction d'une base de données répertoriant les entités et notions du domaine étudié. Cette base de données est présentée sous la forme d'un ou plusieurs fichiers *dspec* qui regroupent l'ensemble des entités du domaine que l'on peut trouver dans les exigences. Cette étape est effectuée en parallèle avec la réécriture des exigences en langage URM. De cette façon, l'utilisateur construit la spécification du

[1] http://www.eclipse.org/modeling/m2t/?project=xpand

domaine au fur et à mesure de la réécriture des exigences. À chaque fois qu'un message ou un argument de message est utilisé dans l'exigence à réécrire, ce dernier l'ajoute dans le fichier DSpec puis le référence dans son exigence URM.

Il est à noter que selon la méthodologie proposée dans ce mémoire, les activités de formalisation des exigences et celle de la formalisation des cas d'utilisation peuvent démarrer indépendamment ou en parallèle, sans un ordre spécifique. De ce fait, on peut réutiliser et potentiellement étendre la spécification du domaine issue de l'activité de spécification des cas d'utilisation si celle-ci a déjà entamé la définition d'une spécification du domaine.

Par la suite, nous utilisons les fragments de patrons de propriétés pour remplacer les termes utiliser dans les exigences par les mots clés listés dans le tableau 7.1. Les exigences issues de ces deux étapes manuelles de réécriture et de consolidation sont regroupées dans le tableau 9.1.

Ainsi, lors de cette phase de réécriture des exigences, pour chacune des exigences, nous avons procédé comme suit:

- Identification des éléments du domaine: Nous avons identifié les éléments appartenant au domaine du composant à valider et nous les avons ajouté à la spécification du domaine produite sous la forme du fichier *SystemManager.dspec*. Ces termes apparaissent dans le tableau 9.1 en soulignés.

- Dans la colonne "Exigences URM", nous avons réécrit les exigences des cahiers des charges en utilisant les mots clés d'affectation et les expressions logiques (AND, OR, WHEN, WITH, =...) présentées dans la section 7.2.1.

- Au cours de la consolidation des exigences, nous utilisons les fragments de patrons proposés dans le tableau 7.1 pour réécrire l'exigence URM. Celle-ci est ensuite analysable par notre algorithme qui détermine quel patron de propriété CDL faut-il instancier pour la génération de la propriété CDL.

La génération des propriétés CDL est obtenue par identification du patron de propriétés à appliquer pour chaque exigence URM. En effet, les exigences URM issues de la phase de consolidation des exigences (section 7.2.2) sont analysées l'algorithme d'analyse des exigences (présenté à la section 7.2.3) afin d'identifier la structure utiliser pour la formalisation de l'exigence. Ainsi, pour chaque exigence, nous avons identifié le patron de spécification de propriété nécessaire à la génération de la propriété CDL. Nous avons utilisé le langage de génération de code *Xpand* avec les templates de propriétés CDL, présentés à la figure 7.8, pour la génération des propriétés CDL.

Table 9.1 : Exigences sur le composant SM du cas d'étude AFS 1/3

ID	Exigence de départ	Exigences URM	Exigence consolidées
473	If the ECDP_DP is in Active_Preparation state, upon reception of a valid EvtRequestMissionPhaseHmi from the ECDP_HMI with operationPhase to OPERATION, the ECDP_DP shall: send an EvtAckHmi with result to TRUE to the ECDP_HMI.	If **STATE**(ECDP_DP) = Active_Preparation, **AND** ECDP_DP **recieves** EvtRequestMissionPhaseHmi **FROM** ECDP_HMI **WITH** operationPhase=OPERATION, **THEN** ECDP_DP **send** EvtAckHmi **WITH** result=TRUE to ECDP_HMI.	It is always the case that If **P** holds THEN **S** eventually holds. **With: P** =[STATE(ECDP_DP) = Active_Preparation & ECDP_DP recieves EvtRequestMissionPhaseHmi FROM ECDP_HMI WITH operationPhase=OPERATION] **S** =[ECDP_DP send EvtAckHmi WITH result=TRUE to ECDP_HMI]
461	If the ECDP_DP is in Active_Preparation state, upon reception of an EvtRequestMissionPhaseHmi with operationPhase to OPERATION, the ECDP_DP shall refuse the state transition if the ECDP_DP is restoring an AFSOperationalConfiguration.	If **STATE**(ECDP_DP) = Active_Preparation, **AND** ECDP_DP **recieves** EvtRequestMissionPhaseHmi **FROM** ECDP_HMI **WITH** operationPhase=OPERATION, **IF** AFSOperationalConfiguration = restoring **THEN** ECDP_DP **shall** refuse the state transition.	It is never the case that **P** holds **IF S** eventually holds. **With: P** =[STATE(ECDP_DP) != Active_Preparation **S** =[AFSOperationalConfiguration=restoring] & ECDP_DP recieves EvtRequestMissionPhaseHmi FROM ECDP_HMI WITH operationPhase=OPERATION]
008	If ECDP_DP is in state Init, upon state change of ECDP_DP to Standby, the ECDP_DP shall send EvtCurrentMissionHmi(IDLE) to each ECDP_HMI in order to set the mission to idle and Send EvtCurrentActivityHmi(LOGIN, true) to each ECDP_HMI in order to activate login	If **STATE**(ECDP_DP) = Init, **AND** ECDP_DP **recieves** StateChangeRequest **FROM** ECDP_HMI **WITH** State=Init, **THEN** ECDP_DP **send** EvtCurrentMissionHmi **WITH** LOGIN=TRUE to **ALL** ECDP_HMI.	It is always the case that If **P** holds THEN **S** eventually holds. **With: P** =[STATE(ECDP_DP) = Init & ECDP_DP recieves StateChangeRequest FROM ECDP_HMI WITH State=Init] **S** =[ECDP_DP send EvtCurrentMissionHmi WITH LOGIN=TRUE to **ALL** ECDP_HMI]

[a] Les termes soulignés représentent des références vers des éléments de la spécification du domaine

Table 9.1 : Exigences sur le composant SM du cas d'étude AFS 2/3 (continued)

ID	Exigence de départ	Exigences URM	Exigence consolidées
020	If the ECDP_DP is in Standby state and if the mission is not yet selected, upon reception of a EvtRequestLoginHmi message from the ECDP_HMI, the ECDP_DP shall send to the ECDP_HMI an EvtAckHmi message with result to TRUE	If **STATE**(ECDP_DP) = Standby, **AND** isMissionSelected = FALSE **AND** ECDP_DP **recieves** EvtRequestLoginHmi **FROM** ECDP_HMI, **THEN** ECDP_DP **send** EvtAckHmi **WITH** result=TRUE to **ALL** ECDP_HMI.	It is always the case that If **P** holds THEN **S** eventually holds. **With:** **P** =[STATE(ECDP_DP) = Standby & Eval(isMissionSelected) = FALSE & ECDP_DP recieves EvtRequestLoginHmi FROM ECDP_HMI] **S** =[ECDP_DP send EvtAckHmi WITH result=TRUE to **ALL** ECDP_HMI]
027	If the ECDP_DP is in active State and if the selected mission is CAP or CDS, upon reception of an EvtRequestLoginHmi message from ECDP_HMI, the ECDP_DP shall validate the password associated to the ECDP_HMI	If **STATE**(ECDP_DP) = active, **AND** isMissionSelected = TRUE **AND** Mission **IN** (CAP, CDS) **AND** ECDP_DP **recieves** EvtRequestLoginHmi **FROM** ECDP_HMI, **THEN** ECDP_DP **send** validateUsername **WITH** to **ALL** ECDP_HMI.	It is always the case that If **P** holds THEN **S** eventually holds. **With:** **P** =[STATE(ECDP_DP) = active & Eval(isMissionSelected) = TRUE (& Eval(Mission) = CAP **OR** Eval(Mission) = CDS) & ECDP_DP recieves EvtRequestLoginHmi FROM ECDP_HMI] **S** =[ECDP_DP send validateUsername to **ALL** ECDP_HMI]
458	Upon reception of an EvtRequestUserPasswordChangeHmi, the ECDP_DP shall change the password associated to the user and send an EvtAckHmi with result to TRUE to the ECDP_HMI.	If ECDP_DP **recieves** EvtRequestUserPasswordChangeHmi **FROM** ECDP_HMI, **THEN** ECDP_DP **send** changePasswordReq **WITH** to **ALL** ECDP_HMI **AND** ECDP_DP **send** EvtAckHmi **WITH** to **ALL** ECDP_HMI **WITH** result = TRUE.	It is always the case that If **P** holds THEN **S** eventually holds. **With:** **P** = ECDP_DP recieves EvtRequestUserPasswordChangeHmi FROM ECDP_HMI] **S** =[ECDP_DP send changePasswordReq to **ALL** ECDP_HMI & ECDP_DP send EvtAckHmi to ALL ECDP_HMI with result = TRUE]

Table 9.1 : Exigences sur le composant SM du cas d'étude AFS 3/3 (continued)

ID	Exigence de départ	Exigences URM	Exigence consolidées
012	Upon reception of a valid EvtRequestLoginHmi from an ECDP_HMI, the ECDP_DP shall send an EvtCommandStatusHmi message with commandId to LOGOUT and status to TRUE to the logged HMI	If ECDP_DP **recieves** EvtRequestLoginHmi **FROM** ECDP_HMI, **THEN** ECDP_DP **send** EvtCommandStatusHmi **WITH** commandId = LOGOUT **AND** status = TRUE to **ALL** ECDP_HMI	It is always the case that If **P** holds THEN **S** eventually holds. **With: P** = ECDP_DP recieves EvtRequestLoginHmi FROM ECDP_HMI] **S** =[ECDP_DP send EvtCommandStatusHmi WITH commandId = LOGOUT & status = TRUE to **ALL** ECDP_HMI]
443	Upon reception of a valid EvtRequestLogoutHmi or upon reception of EvtRequestStopMissionHmi, the ECDP_DP shall send an EvtCommandStatusHmi with commandId to LOGOUT and status to FALSE to the logged ECDP_HMI.	If ECDP_DP **recieves** EvtRequestLogoutHmi **OR** EvtRequestStopMissionHmi **FROM** ECDP_HMI, **THEN** ECDP_DP **send** EvtCommandStatusHmi **WITH** commandId = LOGOUT **AND** status = TRUE to **ALL** ECDP_HMI	It is always the case that If **P** holds THEN **S** eventually holds. **With: P** = ECDP_DP recieves EvtRequestLoginHmi FROM ECDP_HMI] & ECDP_DP recieves EvtRequestStopMissionHmi FROM ECDP_HMI] **S** =[ECDP_DP send EvtCommandStatusHmi WITH commandId = LOGOUT & status = TRUE to **ALL** ECDP_HMI]
016	Upon reception of an EvtRequestMissionHmi message from an ECDP_HMI, the ECDP_DP shall send to all logged ECDP_HMI an EvtAvailableRolesHmi.	If ECDP_DP **recieves** EvtRequestMissionHmi, **THEN** ECDP_DP **send** EvtAvailableRolesHmi	It is always the case that If **P** holds THEN **S** eventually holds. **With: P** = ECDP_DP recieves EvtRequestMissionHmi FROM ECDP_HMI] **S** =[ECDP_DP send EvtAvailableRolesHmi to **ALL** ECDP_HMI]
013	Upon reception of a valid EvtRequestLogoutHmi message from an ECDP_HMI, If there is other logged operators, the ECDP_DP shall release the operator role associated with the ECDP_HMI, reallocate his role to the other logged ECDP_HMI by sending a EvtCurrentRoleHmi.	If ECDP_DP **recieves** EvtRequestLogoutHmi **AND** LoggedOperatorNumber > 0, **THEN** ECDP_DP **send** releaseRole to ECDP_HMI **AND** ECDP_DP **send** EvtCurrentRoleHmi to ECDP_HMI	It is always the case that If **P** holds THEN **S** eventually holds. **With: P** = ECDP_DP recieves EvtRequestLogoutHmi FROM ECDP_HMI] & LoggedOperatorNumber > 0 **S** =[ECDP_DP send releaseRole to ECDP_HMI] & ECDP_DP send EvtCurrentRoleHmi to ECDP_HMI

9.3 Discussion et synthèse

Nous avons présenté une application de la méthodologie de vérification proposée dans ce mémoire sur un cas d'étude industriel de taille et réelle. Tout d'abord, nous avons formalisé les interactions entre le composant à valider et son environnement. La technique proposée est basée sur la formalisation des cas d'utilisation dans des modèles CDL. Ceux-ci décrivent les interactions entre le modèle à valider et son environnement. L'hypothèse forte de cette méthode est que le concepteur est en mesure de spécifier les interactions importantes entre son système et l'environnement dans lequel le système va opérer. Nous justifions cette hypothèse, en particulier dans le domaine de l'embarqué par le fait que le concepteur d'un système doit connaître le périmètre (contraintes, conditions) de son utilisation pour pouvoir le développer correctement. En conséquence, le processus de développement doit inclure une étape de spécification de l'environnement permettant d'identifier un ensemble complet de toutes les interactions entre l'environnement et le modèle, ce qui assurait un taux de couverture de 100%. Dans notre approche, nous avons identifié une activité dédiée à la description des cas d'utilisation étendus afin d'y intégrer la spécification des scénarios d'interaction entre le système et son environnement. Le langage XUC, proposé à cet effet, intègre les constructions nécessaires permettant aux concepteurs de décrire des scénarios d'interactions simples, puis les composer sous la forme de scénarios complexes. Par la suite, cet ensemble de scénarios est traduit en une spécification formelle sous la forme de modèles CDL. De même pour les exigences des cahiers des charges. Celles-ci ont été réécrites en langage URM afin de générer des propriétés CDL. Ces propriétés sont obtenues suite à l'identification des patrons de spécification après l'analyse structurelle des exigences consolidées avec les fragments de propriétés. Ces programmes CDL sont exploités dans un processus de vérification encadré et outillé.

10 Conclusion Générale et Perspectives

10.1 Conclusion

Les méthodes formelles permettent de confronter le modèle de conception aux propriétés à vérifier pour détecter les éventuelles erreurs de conception [Bertolino 2003]. Cependant, ces méthodes ne sont pas applicables directement dans de nombreux contextes industriels pour plusieurs raisons. Parmi ces raisons, on trouve la difficulté de mise en œuvre des outils de vérification existants sur des modèles de taille industrielle. Ceci vient surtout du manque d'adéquation entre les modèles manipulés lors des phases de conception (que nous avons appelés: modèles utilisateurs), et ceux nécessaires à la vérification (modèles CDL).

Ainsi, nous avions pour objectif de réduire l'écart entre les modèles manipulés par les ingénieurs au cours du processus de développement et les modèles CDL. Ceci afin d'améliorer l'intégration des techniques de vérification formelles au processus de développement industriel. Pour répondre à cet objectif, nous avons proposé une méthodologie de vérification basée sur la manipulation de composants UCM. Ces composants visent à construire, de façon incrémentale, des modèles intermédiaires permettant la génération automatique des modèles CDL à partir des artefacts manipulés lors des activités de développement. Notamment en se basant sur les premiers modèles d'exigences et de spécification du système à valider. De cette façon, nous nous sommes imposé les contraintes suivantes lors de la définition de notre méthodologie de vérification: Compatibilité avec les pratiques industrielles courantes, adaptabilité aux processus de développement de logiciels embarqués soumis à des contraintes temporelles ainsi qu'à la complexité des logiciels réels.

La compatibilité avec les pratiques industrielles courantes est assurée par l'adoption des cas d'utilisation largement répandus pour le développement de systèmes logiciels. Nous avons étendu les cas d'utilisation UML, dans un DSL que nous avons appelé XUC, afin d'intégrer la description détaillée des scénarios d'interactions entre le modèle et les acteurs du contexte. Aussi, nous avons proposé de spécifier les exigences sous la forme d'un langage naturel contrôlé (URM). Les exigences sont analysées structurellement afin d'identifier le patron de spécification de propriété nécessaire pour leur formalisation.

Nous avons proposer de spécifier les contextes à un niveau d'abstraction plus élevé, que les simple échanges de messages entre les composants du système, et de générer automa-

tiquement les modèles CDL décrivant les comportements des acteurs de l'environnement pour chaque cas d'utilisation. Ceci dans le but de faciliter la tâche à l'utilisateur en lui permettant de s'appuyer sur les cas d'utilisation qu'il a l'habitude de rédiger. Ainsi, les scénarios sont attachés à chaque cas d'utilisation, et peuvent être enrichis de façon incrémentale au cours du processus de vérification. Les scénarios sont par la suite utilisés pour générer les modèles comportementaux des acteurs de l'environnement afin de pouvoir synthétiser les modèles de contextes à l'aide des algorithmes proposés dans ce mémoire. Nous avons aussi proposé d'utiliser un vocabulaire du domaine afin de lister l'ensemble de ces échanges au sein d'une spécification du domaine à l'aide du langage dédié (DSpec).

Les modèles XUC et URM sont exploités à travers une chaîne de transformations permettant la génération automatique de modèles CDL. Les modèles CDL générés sont directement exploitables par la chaîne d'outil OBP pour évaluer les exigences sur le modèle de conception soumis à validation. Ainsi, les structures UCM regroupent toutes les données et les algorithmes permettant la génération automatiquement des modèles CDL.

Notre approche a été expérimentée sur un cas d'étude industriel réel. L'application de la méthodologie proposée dans ce mémoire a permis la génération de modèles CDL pour évaluer les exigences exprimées dans les cahiers des charges. Ainsi, nous avons étudié la pertinence de notre approche sur le composant `SM` fourni par notre partenaire industriel. Des études complémentaires sont en cours, les ingénieurs qui travaillent sur le projet utilisent nos outils prototypes pour valider le modèle d'autres composants logiciel.

10.2 Perspectives

Ce manuscrit a permis de montrer qu'une approche à base de modèles simples permet de générer rapidement des modèles CDL utilisables pour vérifier des propriétés sur des modèles de taille industrielle. Toutefois, de nombreuses pistes sont à explorer pour permettre d'améliorer l'approche proposée:

Extension du langage naturel contrôlé

Une piste de recherche à suivre pour obtenir une méthode de génération de propriétés plus complète à partir des exigences sera d'étendre le langage URM proposé avec d'avantages de constructions grammaticales pour prendre en charge un plus grand nombre d'exigences. Nous aimerions, en effet, disposer des moyens d'exprimer des contraintes de qualité de service comme le temps de réponse ou les contraintes mémoires de manière à vérifier la qualité de service du produit fini.

Nous avons proposé une approche qui permet d'utiliser un langage proche du langage naturel, mais qui reste néanmoins très contraint: notre langage ne permet d'utiliser qu'un sous-ensemble très restreint de structures grammaticales. L'utilisation des résultats de nombreuses recherches sur l'analyse du langage naturel simplifierait encore l'utilisation

de notre approche pour formaliser un large éventail d'exigences.

Génération de la spécification du domaine

Concernant les modèles du contexte, le langage XUC proposé dans ce mémoire permet de spécifier les scénarios d'interaction entre le composant à valider et les acteurs de l'environnement. Cette spécification est aussi simplifiée grâce au vocabulaire du domaine utiliser. Ce dernier donne aux ingénieurs l'accès aux différents messages et notions pouvant être échangés et permet de vérifier la cohérence des scénarios spécifier. Nous aimerions pouvoir importer d'autres modèles comportementaux directement dans les scénarios des XUC par exemple des diagrammes de séquences. Ceci permettra de faciliter la phase de spécification des scénarios des cas d'utilisation du composant à valider.

Exploitation des données du diagnostic directement des les UCM

Dans la configuration actuelle, OBP délivre à l'utilisateur un retour de preuve lui indiquant si les propriétés à vérifier sont vraies ou fausses. En cas d'échec de la propriété (identifiée comme étant fausse), il lui indique quelle séquence d'exécution de l'environnement (graphe d'exécution) est concernée durant la preuve. Cette indication peut l'aiguiller sur le scénario ayant mis en échec la propriété. Il serait profitable de pouvoir exploiter ces indications et les remonter au niveau des UCM construits afin d'obtenir des facilités identifier les sources d'erreurs dans les modèles de conceptions.

En effet, la synthèse des résultats de preuves, expérimentées dans le cadre du cas d'étude présenté dans ce mémoire, fait apparaître la possibilité de construire un ensemble d'observateurs, spécifiés grâce aux patrons de définition. Cet ensemble d'observateurs permet à l'utilisateur de récupérer un diagnostic à la demande en fonction de l'exigence ou des exigences qu'il veut vérifier sur son modèle. Nous pouvons voir la manipulation de ces ensembles d'observateurs comme un outil de mise au point du modèle.

A Modèles et exigences du cas d'étude

A.1 Listes des exigences du *Session Manager*

Le tableau A.1 regroupe les exigences proposées sur le composant SM du cas d'étude AFS.

Table A.1 : Exigences sur le composant SM du cas d'étude AFS 1/2

ID	Exigence
473	If the ECDP_DP is in Active_Preparation state, upon reception of a valid EvtRequestMissionPhaseHmi from the ECDP_HMI with operationPhase to OPERATION, the ECDP_DP shall: send an EvtAckHmi with result to TRUE to the ECDP_HMI.
461	If the ECDP_DP is in Active_Preparation state, upon reception of an EvtRequestMissionPhaseHmi with operationPhase to OPERATION, the ECDP_DP shall refuse the state transition if the ECDP_DP is restoring an AFSOperationalConfiguration.
008	If ECDP_DP is in state Init, upon state change of ECDP_DP to Standby, the ECDP_DP shall send EvtCurrentMissionHmi(IDLE) to each ECDP_HMI in order to set the mission to idle and Send EvtCurrentActivityHmi(LOGIN, true) to each ECDP_HMI in order to activate login.

Table A.1 : Exigences sur le composant SM du cas d'étude AFS 2/2 (continued)

ID	Exigence
020	If the ECDP_DP is in Standby state and if the mission is not yet selected, upon reception of a EvtRequestLoginHmi message from the ECDP_HMI, the ECDP_DP shall send to the ECDP_HMI an EvtAckHmi message with result to TRUE.
027	If the ECDP_DP is in active State and if the selected mission is CAP or CDS, upon reception of an EvtRequestLoginHmi message from ECDP_HMI, the ECDP_DP shall validate the password associated to the ECDP_HMI.
458	Upon reception of an EvtRequestUserPasswordChangeHmi, the ECDP_DP shall change the password associated to the user and send an EvtAckHmi with result to TRUE to the ECDP_HMI.
012	Upon reception of a valid EvtRequestLoginHmi from an ECDP_HMI, the ECDP_DP shall send an EvtCommandStatusHmi message with commandId to LOGOUT and status to TRUE to the logged HMI.
443	Upon reception of a valid EvtRequestLogoutHmi or upon reception of EvtRequestStopMissionHmi, the ECDP_DP shall send an EvtCommandStatusHmi with commandId to LOGOUT and status to FALSE to the logged ECDP_HMI.
016	Upon reception of an EvtRequestMissionHmi message from an ECDP_HMI, the ECDP_DP shall send to all logged ECDP_HMI an EvtAvailableRolesHmi.
013	Upon reception of a valid EvtRequestLogoutHmi message from an ECDP_HMI, If there is other logged operators, the ECDP_DP shall release the operator role associated with the ECDP_HMI, reallocate his role to the other logged ECDP_HMI by sending a EvtCurrentRoleHmi.

A.2 Liste des événements et interactions CDL

Cette section présente l'ensemble des événements et interactions CDL concernants le cas d'étude *AFS*.

Code CDL

```
event HMI0_r_ack_true {receive SIGNAL_toHMI0_ack_true() from sessionManager};
event HMI0_r_ack_false {receive SIGNAL_toHMI0_ack_false() from sessionManager};

event HMI1_r_ack_true {receive SIGNAL_toHMI1_ack_true() from sessionManager};
event HMI1_r_ack_false {receive SIGNAL_toHMI1_ack_false() from sessionManager};

HMI (login)

event HMI0_s_op0_login {send SIGNAL_fromHMI0_op0_login_true() to sessionManager};
event HMI0_s_op2_login_true {send SIGNAL_fromHMI0_op2_login_true() to sessionManager};
event HMI0_s_op2_login_false {send SIGNAL_fromHMI0_op2_login_false() to sessionManager};

event HMI1_s_op1_login {send SIGNAL_fromHMI1_op1_login_true() to sessionManager};
event HMI1_s_op2_login_true {send SIGNAL_fromHMI1_op2_login_true() to sessionManager};
event HMI1_s_op2_login_false {send SIGNAL_fromHMI1_op2_login_false() to sessionManager};

HMI (logout)

event HMI0_s_logout {send SIGNAL_fromHMI0_logout() to sessionManager};
event HMI1_s_logout {send SIGNAL_fromHMI1_logout() to sessionManager};

HMI (mission)

event HMI0_s_mission_admin {send SIGNAL_fromHMI0_mission_admin() to sessionManager};
event HMI0_s_mission_maintain {send SIGNAL_fromHMI0_mission_maintain() to sessionManager};
event HMI0_s_mission_CDS {send SIGNAL_fromHMI0_mission_CDS() to sessionManager};
event HMI0_s_mission_CAP {send SIGNAL_fromHMI0_mission_CAP() to sessionManager};
event HMI0_s_mission_CRV {send SIGNAL_fromHMI0_mission_CRV() to sessionManager};
event HMI1_s_mission_admin {send SIGNAL_fromHMI1_mission_admin() to sessionManager};
event HMI1_s_mission_maintain {send SIGNAL_fromHMI1_mission_maintain() to sessionManager};
event HMI1_s_mission_CDS {send SIGNAL_fromHMI1_mission_CDS() to sessionManager};
event HMI1_s_mission_CAP {send SIGNAL_fromHMI1_mission_CAP() to sessionManager};
event HMI1_s_mission_CRV {send SIGNAL_fromHMI1_mission_CRV() to sessionManager};

HMI (role selection)

event HMI0_s_role_admin {send SIGNAL_fromHMI0_role_admin() to sessionManager};
event HMI0_s_role_maintain {send SIGNAL_fromHMI0_role_maintain() to sessionManager};
event HMI0_s_role_airPicture {send SIGNAL_fromHMI0_role_airPicture() to sessionManager};
event HMI0_s_role_engagement {send SIGNAL_fromHMI0_role_engagement() to sessionManager};
event HMI1_s_role_admin {send SIGNAL_fromHMI1_role_admin() to sessionManager};
event HMI1_s_role_maintain {send SIGNAL_fromHMI1_role_maintain() to sessionManager};
event HMI1_s_role_airPicture {send SIGNAL_fromHMI1_role_airPicture() to sessionManager};
event HMI1_s_role_engagement {send SIGNAL_fromHMI1_role_engagement() to sessionManager};

HMI (stop mission)

event HMI0_s_stopMission {send SIGNAL_fromHMI0_stopMission() to sessionManager};
event HMI1_s_stopMission {send SIGNAL_fromHMI1_stopMission() to sessionManager};

HMI (activeSubstate)

event HMI0_s_activeSubstate_preparation
                {send SIGNAL_fromHMI0_activeSubstate_preparation() to sessionManager};
event HMI0_s_activeSubstate_operation
                {send SIGNAL_fromHMI0_activeSubstate_operation() to sessionManager};

event HMI1_s_activeSubstate_preparation
                {send SIGNAL_fromHMI1_activeSubstate_preparation() to sessionManager};
```

```
event HMI1_s_activeSubstate_operation
                {send SIGNAL_fromHMI1_activeSubstate_operation() to sessionManager};

HMI (shutdown)

event HMI0_s_shutdown {send SIGNAL_fromHMI0_shutdown() to sessionManager};
event HMI1_s_shutdown {send SIGNAL_fromHMI1_shutdown() to sessionManager};

Acknowledgements from session manager

interaction SM_ack_HMI0 {HMI0_r_ack_true}
interaction SM_nack_HMI0 {HMI0_r_ack_false}

interaction SM_ack_HMI1 {HMI1_r_ack_true}
interaction SM_nack_HMI1 {HMI1_r_ack_false}

HMIx (login)

interaction HMI0_op0_login {HMI0_s_op0_login}
interaction HMI0_op2_login_true {HMI0_s_op2_login_true}
interaction HMI0_op2_login_false {HMI0_s_op2_login_false}

alt HMI0_op2_login {HMI0_op2_login_true, HMI0_op2_login_false}
alt HMI0_any_login {HMI0_op0_login, HMI0_op2_login}

interaction HMI1_op1_login {HMI1_s_op1_login}
interaction HMI1_op2_login_true {HMI1_s_op2_login_true}
interaction HMI1_op2_login_false {HMI1_s_op2_login_false}

alt HMI1_op2_login {HMI1_op2_login_true, HMI1_op2_login_false}
alt HMI1_any_login {HMI1_op1_login, HMI1_op2_login}

HMIx (logout)

interaction HMI0_logout {HMI0_s_logout}
interaction HMI1_logout {HMI1_s_logout}

HMIx (mission)

interaction HMI0_mission_admin {HMI0_s_mission_admin}
interaction HMI0_mission_maintain {HMI0_s_mission_maintain}
interaction HMI0_mission_CDS {HMI0_s_mission_CDS}
interaction HMI0_mission_CAP {HMI0_s_mission_CAP}
interaction HMI0_mission_CRV {HMI0_s_mission_CRV}

alt HMI0_mission_any
                {HMI0_mission_admin, HMI0_mission_maintain,
                 HMI0_mission_CDS, HMI0_mission_CAP, HMI0_mission_CRV}

interaction HMI1_mission_admin {HMI1_s_mission_admin}
interaction HMI1_mission_maintain {HMI1_s_mission_maintain}
interaction HMI1_mission_CDS {HMI1_s_mission_CDS}
interaction HMI1_mission_CAP {HMI1_s_mission_CAP}
interaction HMI1_mission_CRV {HMI1_s_mission_CRV}

alt HMI1_mission_any
                {HMI1_mission_admin, HMI1_mission_maintain,
                 HMI1_mission_CDS, HMI1_mission_CAP, HMI1_mission_CRV}

HMIx (role)

interaction HMI0_role_admin {HMI0_s_role_admin}
interaction HMI0_role_maintain {HMI0_s_role_maintain}
interaction HMI0_role_airPicture {HMI0_s_role_airPicture}
interaction HMI0_role_engagement {HMI0_s_role_engagement}
```

```
125
126 alt HMI0_role_any
127                 {HMI0_role_admin, HMI0_role_maintain,
128                  HMI0_role_airPicture, HMI0_role_engagement}
129
130 interaction HMI1_role_admin {HMI1_s_role_admin}
131 interaction HMI1_role_maintain {HMI1_s_role_maintain}
132 interaction HMI1_role_airPicture {HMI1_s_role_airPicture}
133 interaction HMI1_role_engagement {HMI1_s_role_engagement}
134
135 alt HMI1_role_any
136                 {HMI1_role_admin, HMI1_role_maintain,
137                  HMI1_role_airPicture, HMI1_role_engagement}
138
139 HMIx (stop mission)
140
141 interaction HMI0_stopMission {HMI0_s_stopMission}
142 interaction HMI1_stopMission {HMI1_s_stopMission}
143
144 HMIx (activeSubstate)
145
146 interaction HMI0_activeSubstate_preparation {HMI0_s_activeSubstate_preparation}
147 interaction HMI0_activeSubstate_operation {HMI0_s_activeSubstate_operation}
148 alt HMI0_activeSubstate_any {HMI0_activeSubstate_preparation, HMI0_activeSubstate_operation}
149
150 interaction HMI1_activeSubstate_preparation {HMI1_s_activeSubstate_preparation}
151 interaction HMI1_activeSubstate_operation {HMI1_s_activeSubstate_operation}
152 alt HMI1_activeSubstate_any {HMI1_activeSubstate_preparation, HMI1_activeSubstate_operation}
153
154 HMIx (shutdown)
155
156 interaction HMI0_shutdown {HMI0_s_shutdown}
157 interaction HMI1_shutdown {HMI1_s_shutdown}
```

Bibliography

[Almendros-Jimenez & Iribarn 2005] J Almendros-Jimenez et L Iribarn. *Describing use cases with activity charts.* Metainformatics, vol. 3511, pages 153–167, 2005. 67

[Alur *et al.* 1997] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer et Sriram K. Rajamani. *Partial-Order Reduction in Symbolic State Space Exploration.* In Computer Aided Verification, pages 340–351, 1997. 4

[Alur *et al.* 2000] R Alur, K Etessami et M Yannakakis. *Inference of message sequence charts.* In Proceedings of the International Conference on Software Engineering, pages 304–313, 2000. 70

[Ambriola & Gervasi 2006] Vincenzo Ambriola et Vincenzo Gervasi. *On the Systematic Analysis of Natural Language Requirements with CIRCE.* Automated Software Engineering, vol. 13, no. 1, pages 107–167, Janvier 2006. 82

[Berthomieu *et al.* 2004] B Berthomieu, P O Ribet et F Vernadat. *The tool TINA - Construction of abstract state spaces for Petri Nets and Time Petri Nets.* International Journal of Production Research, vol. 42, no. 14, pages 298–304, 2004. 4, 43, 121

[Berthomieu *et al.* 2007] B Berthomieu, JP Bodeveix et M Filali. *The syntax and semantics of Fiacre.* Rapport technique, 2007. 35

[Berthomieu *et al.* 2008] B Berthomieu, JP Bodeveix, P Farail, M Filali, H Garavel, P Gaufillet, F Lang et F Vernadat. *Fiacre: an Intermediate Language for Model Verification in the Topcased Environment.* Rapport technique, 2008. 35, 121

[Bertolino 2003] Antonia Bertolino. Formal Methods for Software Architectures, volume VII of *978-3-540-20083-3.* Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, 2003. 137

[Booch *et al.* 1999] G Booch, J Rumbaugh et I Jacobson. The Unified Modeling Language User Guide. Addison-Wesley Longman Inc, 1999. 19, 29, 30, 70, 115

[Booch 2004] Grady Booch. Object-oriented analysis and design with applications (3rd edition). Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004. 13

[Bosnacki & Holzmann 2005] Dragan Bosnacki et Gerard J. Holzmann. *Improving Spin's Partial-Order Reduction for Breadth-First Search.* In SPIN, pages 91–105, 2005. 4

[Bozga *et al.* 1999] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier et Joseph Sifakis. *IF: An intermediate representation for SDL and its applications.* In SDL Forum, pages 423–440, 1999. 45, 121

[Brottier 2010] Erwan Brottier. *Brottier09*. PhD thesis, IRISA/TRISKELL IFSIC, 2010. 25

[Broy & Slotosch 1999] Manfred Broy et Oscar Slotosch. *Enriching the Development Process with Formal Methods*. pages 44–61. Springer, 1999. xiii, 31, 32

[Broy & Slotosch 2001] Manfred Broy et Oscar Slotosch. *From Requirements to Validated Embedded Systems*. EMSOFT, vol. 2211, pages 51–65, 2001. 26

[Buhr 1998] R.J.A Buhr. *Use case maps as architectural entities for complex systems*. Software Engineering, IEEE Transactions on, vol. 24, no. 12, pages 1131–1155, 1998. 21

[Carroll 1995] J.M Carroll. Scenario-based design: envisioning work and technology in system development. Wiley, New York 1995. 70

[Cheng 2007] BHC Cheng. *Research directions in requirements engineering*. Future of Software Engineering, ICSE, pages 285–303, 2007. 23, 24, 47, 82

[Cimatti *et al.* 2000] A. Cimatti, E. Clarke, F. Giunchiglia et M. Roveri. *NuSMV: a new symbolic model checker*. Int. J. on Software Tools for Technology Transfer, vol. 2, no. 4, pages 410–425, 2000. 4

[Clarke *et al.* 1986] E.M. Clarke, E.A. Emerson et A.P. Sistla. *Automatic verification of finite-state concurrent systems using temporal logic specifications*. ACM Trans. Program. Lang. Syst., vol. 8, no. 2, pages 244–263, 1986. 4

[Cockburn 2000] Alistair Cockburn. Writing effective use cases. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st édition, 2000. 19, 22, 67, 76, 77

[Colombo *et al.* 2007] P Colombo, V Del Bianco, L Lavazza et A Coen. *A methodological framework for SysML: a Problem Frames-based approach*. Software Engineering Conference, 2007. 115

[Craigen *et al.* 1995] D. Craigen, S. Gerhart et T. Ralston. *Formal methods reality check: industrial usage*. IEEE Transactions On Software Engineering, vol. 21, no. 2, pages 90–98, 1995. 24

[Dalal A & Uchitel 2007] Russo ORA Dalal A et Sebastian Uchitel. *Extracting Requirements from Scenarios with ILP*. Inductive Logic Programming, vol. LNAI 4455, pages 1–15, Mai 2007. 63

[Darimont & van Lamsweerde 1996] R Darimont et Axel van Lamsweerde. *Formal refinement patterns for goal-driven requirements elaboration*. ACM SIGSOFT Software Engineering Notes, vol. 21, no. 6, page 190, 1996. 82

[Davis 1993] A.M. Davis. *Software requirements: objects, functions and states*. Uppper Saddle River, NJ, USA, 1993. Prentice-Hall. 3

BIBLIOGRAPHY

[Denger *et al.* 2003] C Denger, DM Berry et E Kamsties. *Higher quality requirements specifications through natural language patterns.* Software: Science, Technology and Engineering, 2003. SwSTE'03. Proceedings. IEEE International Conference on, pages 80–90, 2003. 25

[Dhaussy & Boniol 2007] Philippe Dhaussy et Frédéric Boniol. *Mise en œuvre de composants MDA pour la validation formelle de modèles de systèmes d'information embarqués.* Revue des Sciences et Techniques Informatiques, pages 133–157, 2007. 5, 37

[Dhaussy & Roger 2011] Philippe Dhaussy et Jean-Charles Roger. *CDL (Context Description Language) : Syntax and semantics.* Rapport technique, ENSTA-Bretagne, 2011. 37

[Dhaussy *et al.* 2009] Philippe Dhaussy, Pierre-Yves Pillain, Stephen Creff, Amine RAJI, Yves Le Traon et Benoit Baudry. *Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation.* ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems, vol. LNCS 5795, pages 438–452, 2009. 5, 37, 40, 43, 63, 66, 121

[Dhaussy *et al.* 2011] Philippe Dhaussy, Frédéric Boniol et Jean-Charles Roger. *Reducing State Explosion with Context Modeling for Model-Checking.* In 13th IEEE International High Assurance Systems Engineering Symposium (Hase'11), Boca Raton, USA, 2011. 5, 37, 38, 39, 40, 44

[Dwyer *et al.* 1999] Matthew B. Dwyer, George S. Avrunin et James C. Corbett. *Patterns in property specifications for finite-state verification.* In 21st Int. Conf. on Software Engineering, pages 411–420. IEEE Computer Society Press, 1999. xiii, 27, 28, 39, 82, 106, 109

[E.A. Emerson and S. Jha and D. Peled 1997] E.A. Emerson and S. Jha and D. Peled. *Combining Partial Order and Symmetry Reductions.* In E. Brinksma, editeur, Tools and Algorithms for the Construction and Analysis of Systems, pages 19–34, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217. 4

[Ebner & Kaindl 2002] G Ebner et H Kaindl. *Tracing all around in reengineering.* IEEE Software, vol. 19, no. 3, pages 70–77, Mai 2002. 13, 14

[Edwards *et al.* 1995] ML Edwards, M Flanzer, M Terry et J Landa. *RECAP: a requirements elicitation, capture and analysis process prototype tool for large complex systems.* IEEE International Conference on Engineering of Complex Computer Systems, Ft. Lauderdale, USA, November, pages 6–10, 1995. 53

[Fabbrini *et al.* 2002] F Fabbrini, M Fusani, S Gnesi et G Lami. *The linguistic approach to the natural language requirements quality: benefit of the use of an automatic tool.* Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard, pages 97–105, 2002. 25, 82

[Fantechi *et al.* 1994] A Fantechi, S Gnesi, G Ristori, M Carenini, M Vanocchi et P Moreschini. *Assisting requirement formalization by means of natural language translation.* Formal Methods in System Design, vol. 4, no. 3, Mai 1994. 82

[Farail *et al.* 2006] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut et Marc Pantel. *The TOPCASED project : a Toolkit in OPen source for Critical Aeronautic SystEms Design.* In 3rd European Congress EMBEDDED REAL TIME SOFTWARE (ERTS), Toulouse, France 2006. 40

[Farail *et al.* 2008] Patrick Farail, Pierre Gaufillet, Florent Peres, Jean-Paul Bodeveix, Mamoun Filali, Bernard Berthomieu, Saad Rodrigo, Francois Vernadat, Hubert Garavel et Frédéric Lang. *FIACRE: an intermediate language for model verification in the TOPCASED environment.* In European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008. SEE, janvier 2008. 43

[Feiler & Humphrey 1993] P. H. Feiler et W. S. Humphrey. *Software process development and enactment: concepts and definitions.* Software Process, 1993. Continuous Software Process Improvement, Second International Conference on the, pages 28–40, Février 1993. 29

[Fellbaum 1998] Christiane Fellbaum. *WordNet: An Electronic Lexical Database.* In Cambridge, MA: MIT Press., 1998. 58

[Fernandez *et al.* 1996] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu et Mihaela Sighireanu. *CADP: A Protocol Validation and Verification Toolbox.* In CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification, pages 437–440, London, UK, 1996. Springer-Verlag. 4

[Ferreira & Rodrigues da Silva 2009] David De Almeida Ferreira et Alberto Rodrigues da Silva. *A Controlled Natural Language Approach for Integrating Requirements and Model-Driven Engineering.* pages 518–523, 2009. 26, 27

[Finkelstein & Kramer 2000] Anthony Finkelstein et Jeff Kramer. Software engineering: a roadmap. ACM, New York, New York, USA, 2000. 24

[Fontan 2008] Benjamin Fontan. *Méthodologie De Conception De Sysèmes Temps Réel Et Distribués En Contexte UML/SysML.* PhD thesis, Université Toulouse III Paul Sabatier, 2008. 31, 117

[Fuchs *et al.* 1999] N Fuchs, U Schwertel et Rolf Schwitter. *Attempto Controlled English—not just another logic specification language.* Logic-Based Program Synthesis and Transformation, vol. 1559/1999, pages 1–20, 1999. 82

[Genest & Muscholl 2005] B Genest et A Muscholl. *Message sequence charts: a survey.* In IEEE, editeur, Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on, numéro 8598201, pages 2–4, 2005. 70

BIBLIOGRAPHY

[Godefroid *et al.* 1996] Patrice Godefroid, Doron Peled et Mark G. Staskauskas. *Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs.* In International Symposium on Software Testing and Analysis, pages 261–269, 1996. 4

[Gutiérrez *et al.* 2008] J Gutiérrez, C Nebut, M Escalona et M Mejías. *Visualization of use cases through automatically generated activity diagrams.* MoDELS, vol. 5301, pages 83–96, 2008. 97

[Harel & Kugler 1999] D. Harel et H. Kugler. *Synthesizing State-Based Object Systems from LSC Specifications.* Rapport technique, Jerusalem, 1999. 70

[Hassine *et al.* 2010] Jameleddine Hassine, J Rilling et R Dssouli. *An evaluation of timed scenario notations.* Journal of Systems and Software, vol. 83, no. 2, pages 326–350, 2010. 25

[Hassine 2005] Hassine. *An ASM operational semantics for use case maps.* In Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on, pages 467–468, 2005. 22

[Haugen *et al.* 2005] Oystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde et Ketil Stolen. *STAIRS towards formal design with sequence diagrams.* Software and System Modeling, vol. 4, no. 4, pages 355–357, 2005. 40

[Heitmeyer 1998] C Heitmeyer. *On the need for practical formal methods.* Formal Techniques in Real-Time and Fault-Tolerant Systems, Proc., 5th Intern. Symposium, vol. Springer Verlag, no. 18–26, 1998. 25

[Heymer 2000] S Heymer. *A semantics for MSC based on petri net components.* 4th International SDL and MSC Workshop (SAM'00), pages 1–15, 2000. 70

[Holzmann 1997] G.J. Holzmann. *The Model Checker SPIN.* Software Engineering, vol. 23, no. 5, pages 279–295, 1997. 4

[Holzmann 2004] G Holzmann. *The SPIN Model Checker, Primer and Reference Manual.* Massachusetts, Massachusetts, 2004. 29

[ITU-TS 1999] ITU-TS. *Recommendation Z.120(11/99) : MSC 2000.* Geneva, 1999. 21

[ITU 1996] ITU. *Message Sequence Chart (MSC).* In ITU-T Recommendation Z.120, Geneva, 1996. 41

[Jac 2000] *Total correctness refinement for sequential reactive systems.* 13th International Conference on Theorem Proving in Higher Order Logics, vol. LNCS, no. 1869, pages 320–337, 2000. 76, 77

[Jackson 2000] Michael Jackson. Problem frames: analyzing and structuring software development problems. Addison-Wesley Longman Publishing Co., Inc., 2000. 24

[Jacobson 2004] Ivar Jacobson. *Use cases Yesterday, today, and tomorrow.* Software and Systems Modeling, vol. 3, pages 210–220, 2004. 23, 67

[Kaindl 1997] Hermann Kaindl. *A practical approach to combining requirements definition and object-oriented analysis* . Annals of Software Engineering, vol. 3, pages 319–343, 1997. 13, 17, 52

[Kaindl 1999] H Kaindl. *Difficulties in the transition from OO analysis to design.* IEEE Software, vol. 16, no. 5, pages 94–102, 1999. 13

[Kof 2004] Leonid Kof. Natural Language Processing to Requirements Engineering: Applicability to large requirements documents. Citeseer, 2004. 53

[Konrad & Cheng 2005a] S. Konrad et B. Cheng. *Real-Time Specification Patterns.* In 27th Int. Conf. on Software Engineering (ICSE05), St Louis, MO, USA, 2005. xv, 7, 28, 39, 82, 107

[Konrad & Cheng 2005b] Sascha Konrad et Betty H C Cheng. *Real-time Specification Patterns.* In International Conference on Software Engineering, pages 372–381, 2005. 106

[Konrad & H C Cheng 2005] Sascha Konrad et Betty H C Cheng. *Facilitating the Construction of Specification Pattern-based Properties.* In IEEE International Requirements Engineering Conference, Paris, France, 2005. 24

[Konrad & H C Cheng 2006] Sascha Konrad et Betty H C Cheng. *Automated Analysis of Natural Language Properties for UML Models.* In International Conference on Model Driven Engineering Languages and Systems, pages 48–57, 2006. xiii, 28, 29, 30

[Koskimies & MLkinen 1994] K Koskimies et E MLkinen. *Automatic synthesis of state machines from trace diagrams.* Software Practice and Experience, vol. 4, pages 643–658, 1994. 70

[Ladkin & Leue 1995] Peter B Ladkin et Stefan Leue. *Interpreting Message Flow Graphs.* Formal Aspects of Computing, vol. 7, no. 5, pages 473–509, Septembre 1995. 70, 82

[Lamsweerde 2009] A. Van Lamsweerde. *Requirements Engineering: From system goals toUML Models to Software Specification.* Wiley Desktop Edition, 712 pages, 2009. 47

[Larsen *et al.* 1997] Kim Guldstrand Larsen, Paul Pettersson et Wang Yi. *UPPAAL in a Nutshell.* International Journal on Software Tools for Technology Transfer, vol. 1, no. 1-2, pages 134–152, 1997. 4

[Lee *et al.* 2007] Insup Lee, Joseph Y-T. Leung et Sang H. Son. Handbook of real-time and embedded systems. Chapman & Hall/CRC, 1st édition, 2007. 3

[Lu *et al.* 2008] Chih-Wei Lu, Chih-Hung Chang, William C. Chu, Ya-Wen Cheng et Hsin-Chien Chang. *A Requirement Tool to Support Model-Based Requirement Engineering.* In Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC '08, pages 712–717, Washington, DC, USA, 2008. IEEE Computer Society. 4

BIBLIOGRAPHY

[Mc.Millan & Probst 1992] K. L. Mc.Millan et D. K. Probst. *A Technique of State Space Search Based on Unfolding.* In Formal Methods in System Design, pages 45–65, 1992. 4

[Michal 2005] Smialek Michal. *Accommodating Informality with Necessary Precision in Use Case Scenarios.* Journal of Object Technology, vol. 4, pages 59–67, 2005. 20

[Miller 1995] George A Miller. *WordNet: A Lexical Database for English.* In Communications of the ACM, pages 39–41, 1995. 58

[Mullery 1979] G.P. Mullery. *Core-A method for controlled requirements specification.* In 4th International Conference on Software Engineering, pages 126–125, London, 1979. The Institution of Eletrical Engineers. 24

[Mustafiz *et al.* 2008] Sadaf Mustafiz, Jörg Kienzle et Andrey Berlizev. *Addressing degraded service outcomes and exceptional modes of operation in behavioural models.* In SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems. ACM, Novembre 2008. 67

[Nikora & Balcom 2009] Allen P Nikora et Galen Balcom. *Automated Identification of LTL Patterns in Natural Language Requirements.* Software Reliability Engineering, International Symposium on, vol. 0, pages 185–194, 2009. 24

[Nuseibeh & Easterbrook 2000] Bashar Nuseibeh et Steve Easterbrook. *Requirements Engineering: A Roadmap.* International Conference on Software Engineering, 2000. 23, 81

[OMG 2007] OMG. *UML 2.1.2 Superstructure.* pages 1–738, 2007. 20, 30, 67, 74, 76, 77, 99, 117

[OMG 2008a] OMG. *Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification.* Rapport technique formal/2008-04-03, Object Management Groupe, http://www.omg.org/spec/QVT/1.0/PDF, avril 2008. 130

[OMG 2008b] OMG. *OMG Systems Modeling Language (OMG SysML^{TM}).* pages 1–256, 2008. 23

[Övergaard & Palmkvist 2004] G Övergaard et K Palmkvist. *A formal approach to use cases and their relationships.* The Unified Modeling Language UML'98: Beyond the Notation, pages 514–514, 2004. 20, 67

[Park & Kwon 2006] S. Park et G. Kwon. *Avoidance of state explosion using dependency analysis in model checking control flow model.* LNCS, 2006. 4

[Peled 1998] D. Peled. *Ten Years of Partial Order Reduction.* In CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification, pages 17–28. Springer-Verlag, 1998. 4

[Pnueli 1977] Amir Pnueli. *The temporal logic of programs.* In SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. 4

[Queille & Sifakis 1982] Jean-Pierre Queille et Joseph Sifakis. *Specification and verification of concurrent systems in CESAR.* In Proceedings of the 5th Colloquium on International Symposium on Programming, pages 337–351, London, UK, 1982. Springer-Verlag. 4

[Raji & Dhaussy 2010a] Amine Raji et Philipe Dhaussy. *Automatic Formal Model Derivation from Use Cases.* In 6èmes journées sur l'Ingénierie Dirigée par les Modèles, pages 1–8, Pau, France, March 2010 2010. Université de Pau et des Pays de l'Adour (UPPA). xvii

[Raji & Dhaussy 2010b] Amine Raji et Philippe Dhaussy. *User Context Models : A Framework to Ease Software Formal Verifications.* In 12th International Conference on Enterprise Information Systems, volume 3, pages 380–383, ISAS, Funchal, Madeira Portugal, June 8 - 12 2010. xvii

[Raji & Dhaussy 2011a] Amine Raji et Philipe Dhaussy. *Modèles orientés utilisateurs pour la vérification formelle en contexte industriel.* In Ileana Ober, editeur, 7ème journées sur l'Ingénierie Dirigée par les Modèles (IDM), volume 1, pages 135–140, Lille, France, Juin 2011. xvii

[Raji & Dhaussy 2011b] Amine Raji et Philippe Dhaussy. *Improving formal verification practicability through user oriented models and context-awareness.* In Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation, volume 4 of *MoDeVVa*, pages 51–54, New York, NY, USA, 2011. ACM. xvii

[Raji & Dhaussy 2011c] Amine Raji et Philippe Dhaussy. *Use Cases Modeling for Scalable Model-Checking.* In Karl Leung Tran Dan Thu, editeur, The 18th Asian Pacific Software Engineering Conference (APSEC), volume 1, pages 65–72. University of Science, VNU-HCM, IEEE Computer Society, December 2011. xvii

[Raji *et al.* 2010] Amine Raji, Philippe Dhaussy et Bruno Aizier. *Automating Context Description for Software Formal Verification.* In MODEVVA '10: Proceedings of the Workshop on Model-Driven Engineering, Verification, and Validation, numéro 978-0-7695-4384-0, pages 76–82. IEEE Computer Society, Octobre 2010. xvii

[Reniers 1999] M A Reniers. *Message Sequence Chart. Syntax and Semantics.* PhD thesis, Eindhoven University of Technology, 1999. 70

[Richards 2003] Debbie Richards. *Merging individual conceptual models of requirements.* Requirements Eng, vol. 8, pages 195–205, 2003. 51

[Roger 2006] Jean-Charles Roger. *Exploitation de contextes et d'observateurs pour la validation formelle de modËles.* PhD thesis, ENST-Bretagne, Université de Rennes I, 2006. 5, 18, 37, 45

[Rosenblum 1996] DS Rosenblum. *Formal methods and testing: why the state-of-the art is not the state-of-the practice.* ACM SIGSOFT Software Engineering Notes, vol. 21, Issue 4, 1996. 24

BIBLIOGRAPHY

[Roychoudhury 2009] Abhik Roychoudhury. Embedded Systems and Software Validation. 978-0-12-374230-8, 272 pages. 2009. 3

[Ryan 1992] K Ryan. *The role of natural language in requirements engineering.* Requirements Engineering, 1993., Proceedings of IEEE International Symposium on, pages 240–242, 1992. 25, 81

[Shui *et al.* 2005] A Shui, Sadaf Mustafiz, J Kienzle et C Dony. *Exceptional use cases.* 8th International Conference on Model Driven Engineering Languages and Systems, vol. LNCS 3713, pages 568–583, 2005. 67

[Shui *et al.* 2006] A Shui, S. Mustafiz et Jörg Kienzle. *Exception-aware requirements elicitation with use cases.* Advanced Topics in Exception Handling, vol. LNCS 4119, pages 221–242, 2006. 67

[Smith *et al.* 2002] R. Smith, G.S. Avrunin, L. Clarke et L. Osterweil. *Propel: An Approach Supporting Property Elucidation.* In 24st Int. Conf. on Software Engineering(ICSE02), St Louis, MO, USA, pages 11–21. ACM Press, 2002. 39

[Sommerville & Sawyer 1997] Ian Sommerville et Pete Sawyer. *Viewpoints: principles, problems and a practical approach to requirements engineering.* Annals of Software Engineering, vol. 3, pages 101–130, 1997. 24

[Stevens 2001] Perdita Stevens. *On Use Cases and Their Relationships in the Unified Modelling Language.* FASE, vol. LNCS 2029, pages 140–155, 2001. 20

[Stevens 2007] Perdita Stevens. *On use cases and their relationships in the Unified Modelling Language.* FASE '01 Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering, pages 140–155, 2007. 20, 76, 77

[Texel & Williams 1997] Putnan P Texel et Charles B Williams. Use cases combined with BOOCH/OMT/UML: Process and products, volume 978-0137274055, 465 pages. Prentice Hall; 1 edition, 1997. 23

[Uchitel *et al.* 2004] Sebastian Uchitel, Jeff Kramer et Jeff Magee. *Incremental elaboration of scenario-based specifications and behavior models using implied scenarios.* ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 13, no. 1, pages 37–85, Janvier 2004. 23, 63

[Uchitel 2003] Sebastian Uchitel. *Incremental Elaboration of Scenario- Based Specifications and Behaviour Models Using Implied Scenarios.* PhD thesis, Imperial College of Science, Technology and Medicine University of London Imperial College of Science, Technology and Medicine, University of London, Department of Computing, Février 2003. 23, 70

[Valmari 1991] Antti Valmari. *Stubborn sets for reduced state space generation.* In Proceedings of the 10th International Conference on Applications and Theory of Petri Nets, pages 491–515, London, UK, 1991. Springer-Verlag. 4

[van Lamsweerde & Letier 2004] Axel van Lamsweerde et E Letier. *From object orientation to goal orientation: A paradigm shift for requirements Engineering.* Radical Innovations of Software & System Engineering, Venice(Italy), LNCS, pages 4–8, 2004. 23

[van Lamsweerde 2000] Axel van Lamsweerde. Requirements engineering in the year 00: a research perspective. a research perspective. 22nd international conference on Software engineering, New York, New York, USA, 2000. 23

[van Lamsweerde 2008] Axel van Lamsweerde. *Requirements engineering: from craft to discipline.* Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pages 238–249, 2008. 23

[Videira & Rodrigues da Silva 2005] Carlos Videira et Alberto Rodrigues da Silva. *Patterns and metamodel for a natural-language-based requirements specification language.* In CaiSE Forum, 2005. 25, 26, 27

[Videira *et al.* 2005] Carlos Videira, Jo Leonardo Carmo et Alberto Rodrigues da Silva. *The ProjectIT-RSL Language Overview.* Satellite Activities, vol. 3297, pages 269–272, 2005. 26

[Whittle & Jayaraman 2006] Jon Whittle et PK Jayaraman. *Generating Hierarchical State Machines from Use Case Charts.* Proceedings, pages 19–28, 2006. 70

[Whittle 2006] Jon Whittle. *Specifying precise use cases with use case charts.* In MoDELS'06, Satellite Events, pages 290–301, 2006. 40

[Whittle 2007] Jon Whittle. *Precise specification of use case scenarios.* FASE'07 Proceedings of the 10th international conference on Fundamental approaches to software engineering, pages 170–184, 2007. 19, 20, 21

[Williams *et al.* 2005] C Williams, M Kaplan et T Klinger. *Toward engineered, useful use cases.* Journal of Object Technology, vol. 4, pages 45–57, 2005. 20

[Wolter *et al.* 2008] Katharina Wolter, Michal Smialek, Daniel Bildhauer et Hermann Kaindl. *Reusing Terminology for Requirements Specifications from WordNet.* In 16th IEEE International Requirements Engineering Conference (RE), pages 325–326, 2008. 52, 58

[Yue *et al.* 2010] T. Yue, L. Briand et Y. Labiche. *A systematic review of transformation approaches between User requirements and analysis models.* Requirements Engineering Journal, vol. 16, no. 2, pages 75–99, 2010. 47

[Zhu & Jin 2002] Hong Zhu et L Jin. *Automating scenario-driven structured requirements engineering.* Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International, pages 311–316, 2002. 75

[Zhu *et al.* 2002] Hong Zhu, L Jin, D Diaper et G Bai. *Software requirements validation via task analysis.* Journal of Systems and Software, vol. 61, no. 2, pages 145–169, 2002. 75

[Ziadi *et al.* 2009] Tewfik Ziadi, Xavier Blanc et Amine RAJI. *From Requirements to Code Revisited*. In ISORC '09: Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pages 228–235. IEEE Computer Society, Mars 2009. 70

Technopole Brest-Iroise - CS 83818
29238 Brest Cedex 3
France
Tel : + 33 (0)2 29 00 11 11
www.telecom-bretagne.eu