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 seront utilisés 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 inclus dans les dictionnaires de vocabulaire pour être tracés 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 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 de 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 que 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 mélangées [Edwards *et al.* 1995]. De ce fait, les significations 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 meilleure 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 list of registred consoles*
- *SM register is ready to receive login request from reachable consoles*
- *Registred console 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 identifiés. 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 "*login 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 étudié 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 définitions 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".

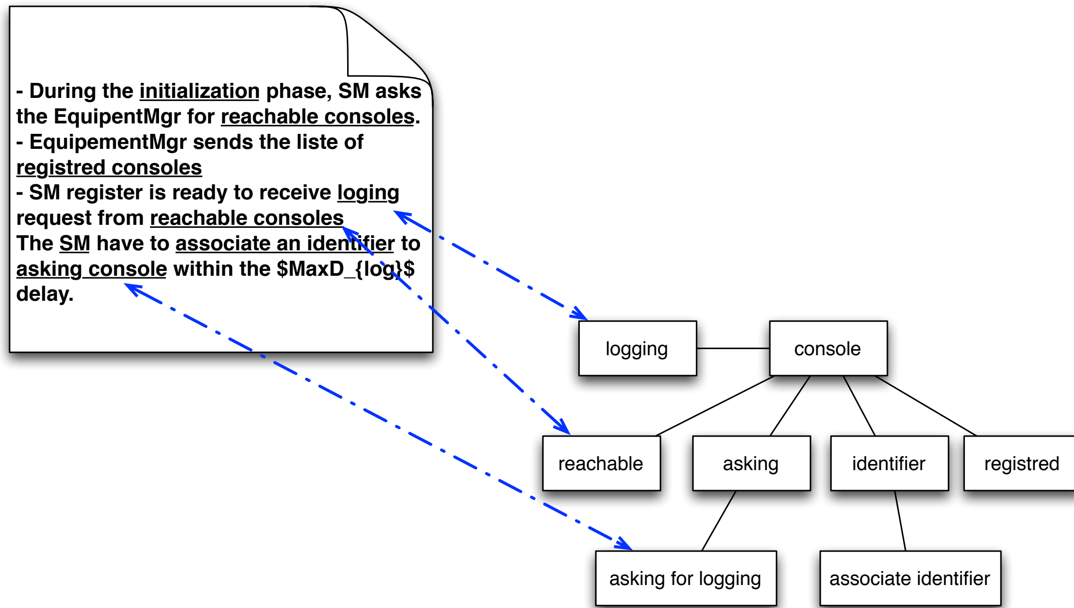

Figure 4.1 : Exigences avec des liens vers le vocabulaire du domaine

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 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 *DSpec* est organisé 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

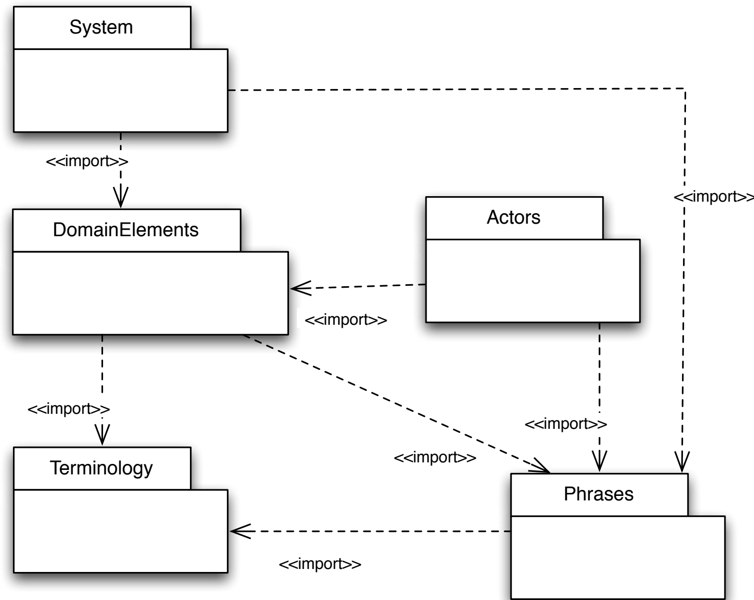

Figure 4.2 : Organisation du métamodèle de spécification du domaine

références vers les termes proposés par le thesaurus.

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.

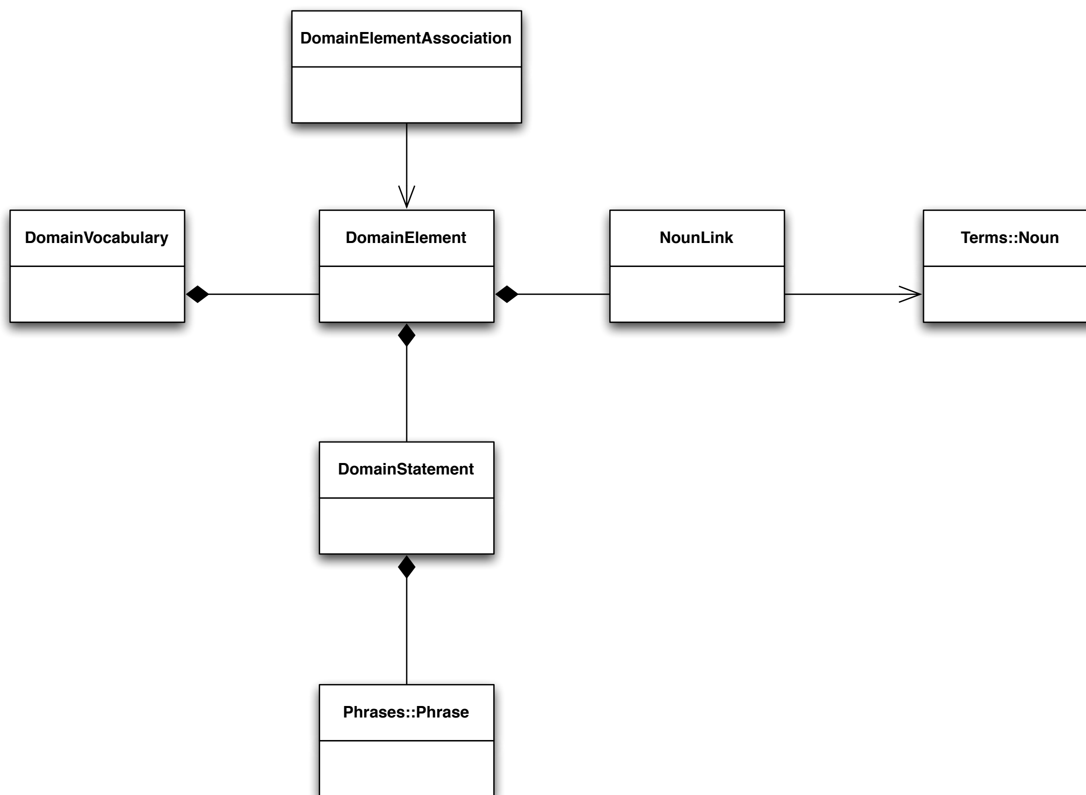

Figure 4.3 : Métamodèle des éléments du domaine et liens entre les packages

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*.

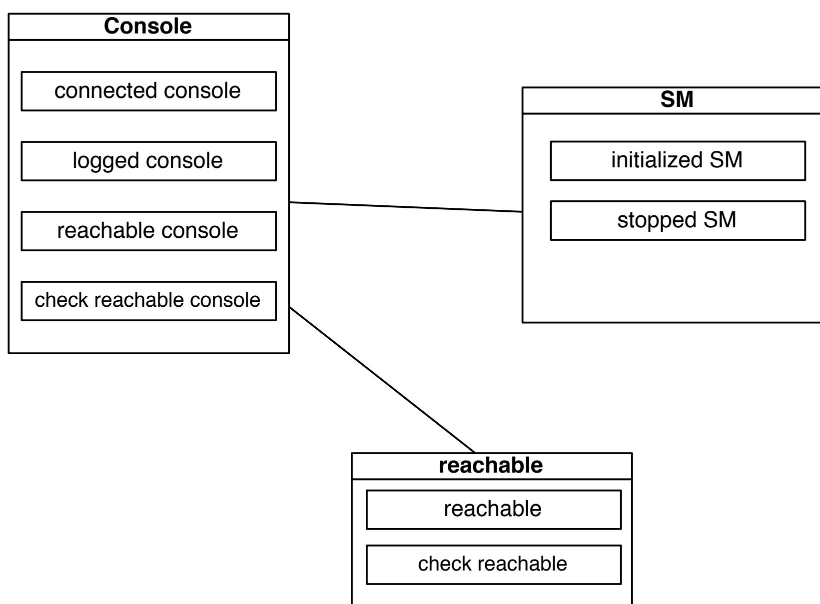

Figure 4.4 : Vue partielle des éléments du domaine de l'*AFS*

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 *Terminology*. 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.

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

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

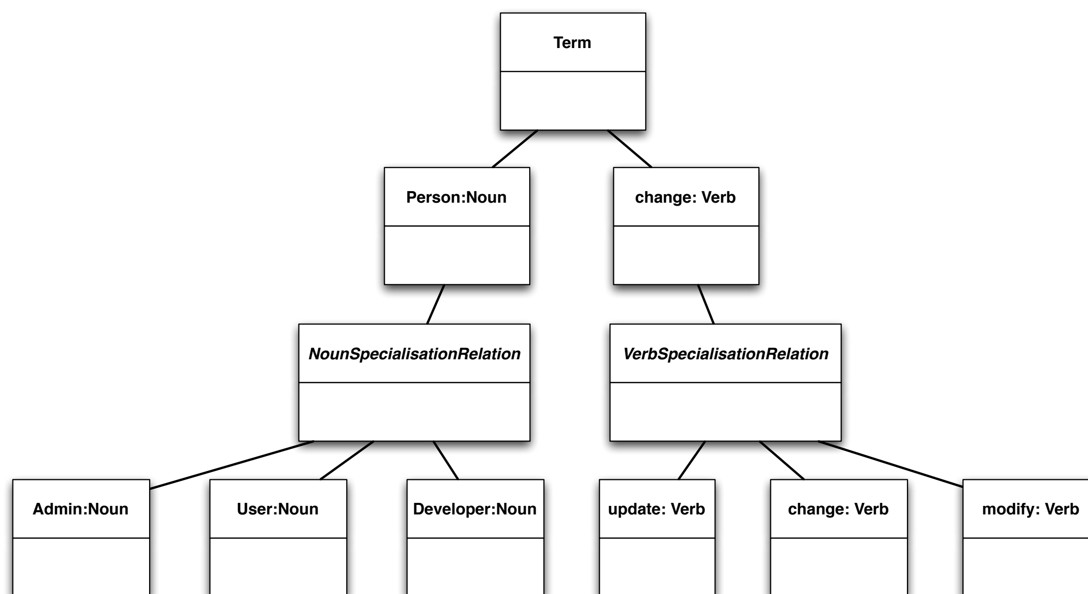

Figure 4.5 : Exemple de relations de spécialisation entre les termes

et adverbes de la langue anglaise [Miller 1995, Fellbaum 1998, Wolter *et al.* 2008]. Les 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

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

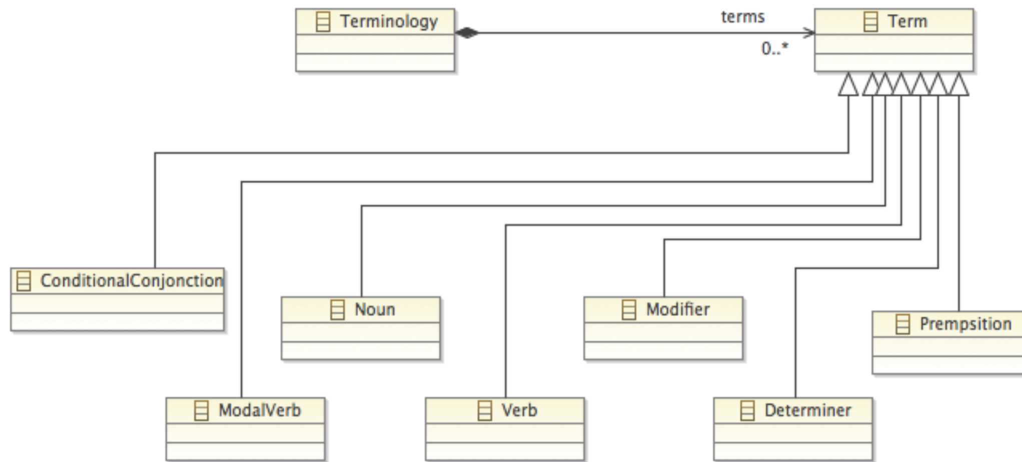

Figure 4.6 : Présentation du package *Terminology*

autres termes (homonymes, synonymes) et les classes selon leurs natures grammaticales. 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 référencer plus tard dans les modèles de contextes et les exigences.

Dans ce chapitre, nous avons présenté un formalisme 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 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

5

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

```
1  CDL Initialisation
2  {
3    events{
4      //Déclaration des événements: Voir l'annexe~A.2
5    }
6    par MAIN{
7      HMI1,
8      HMI2,
9      EquipMgr
10   }
11
12   //Déclaration des interactions
13   interaction HMI1_init_op1          { HMI1_s_initConsolAck_OK }
14   interaction HMI2_init_op1          { HMI2_s_initConsolAck_OK }
15   interaction HMI1_init_op2          { HMI1_s_initConsolAck_KO }
16   interaction HMI2_init_op2          { HMI2_s_initConsolAck_KO }
17   interaction EquiMgr_Init_op1 {EquiMgr_s_goInitAck_OK, EquiMgr_s_initConsol,
18                                 EquiMgr_s_addConsole}
19   interaction EquiMgr_Init_op2 {EquiMgr_s_goInitAck_KO}
20   interaction EquiMgr_Init_op3 {EquiMgr_r_addConsolAck_KO}
21   interaction EquiMgr_Init_op4 {EquiMgr_r_addConsolAck_OK}
22
23   alt HMI1
24     {
25           HMI1_init_op1,
26           HMI1_init_op2,
27     }
28   alt HMI2
29     {
30           HMI2_init_op1,
31           HMI2_init_op2,
32     }
33   alt EquiMgr
34     {
35       EquiMgr_Init_op1,
36       EquiMgr_Init_op2,
37       EquiMgr_Init_op3,
38       EquiMgr_Init_op4
39     }
40 } // 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].

5

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 suivants :

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 types 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 influencer sur 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

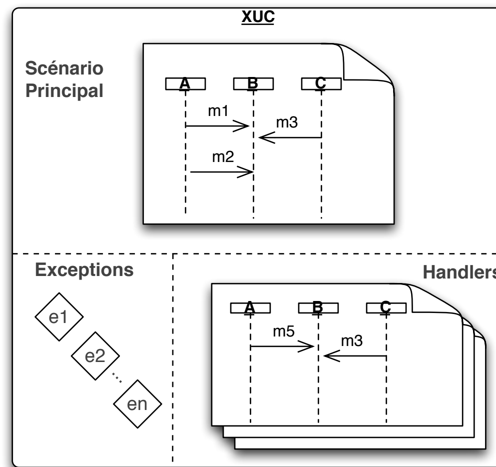

Figure 5.3 : Structuration des scénarios au sein d'un XUC

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

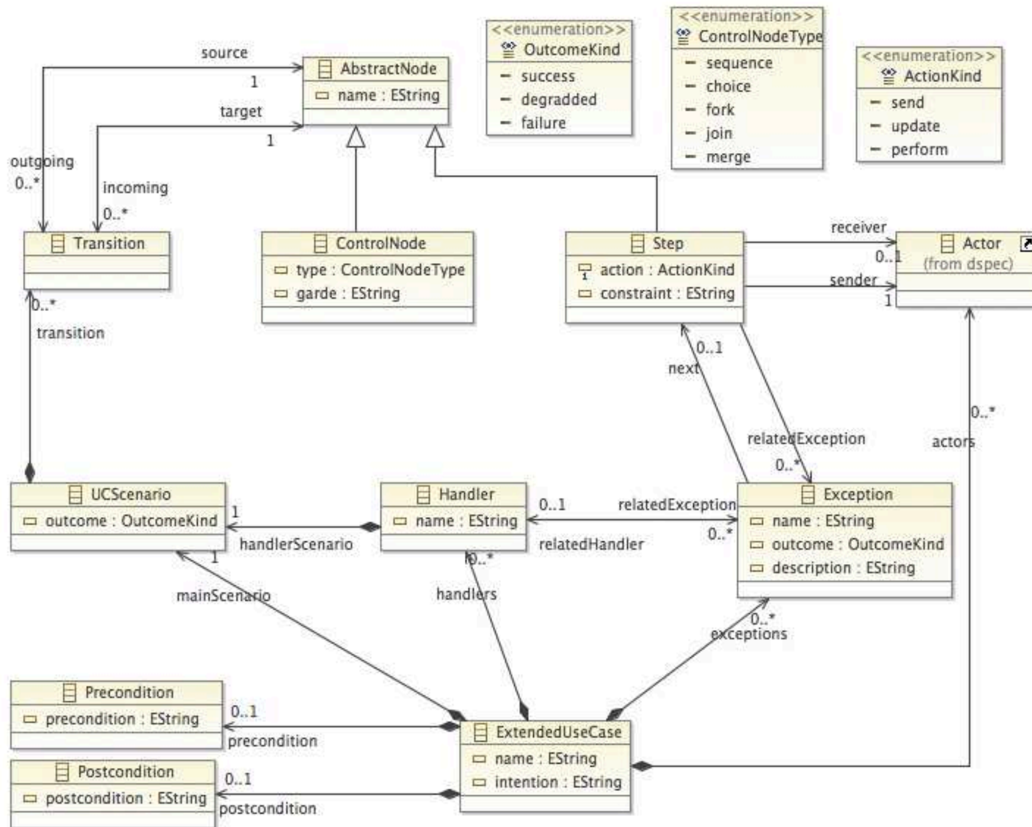

Figure 5.4 : Métamodèle des XUC

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

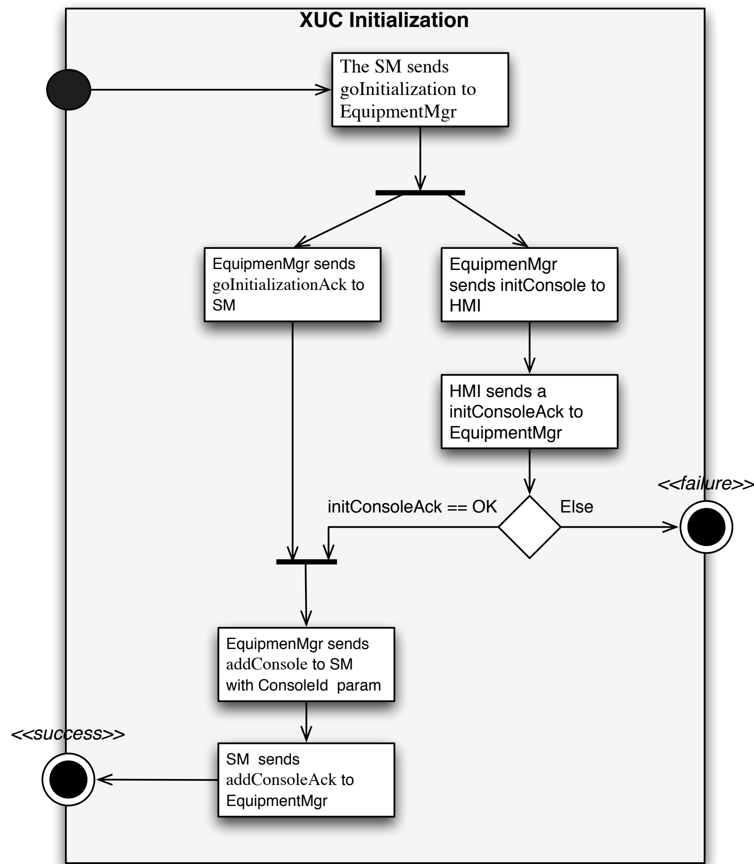

Figure 5.5 : Représentation graphique du scénario d'initialisation

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 $\lambda = T \times St \cup Ps \times T$.
- 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 $s \in St$ 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*:$(k, m) \in \{!, ?\} \times M$.
- $Ps = S_0 \cup EP \cup Fk \cup Jn \cup Ch, Sf$ 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 son 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`).

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

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 $N \longrightarrow N'$.

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

5

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.

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

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 $N \longrightarrow N'$.

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

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.

$$\frac{\overline{l : decision(\langle g \rangle N_1, \langle \neg g \rangle N_2)}}{l : decision(\langle False \rangle N_1, \langle True \rangle \bar{N}_2)} \quad \text{[decision1]}$$

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

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)

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

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].

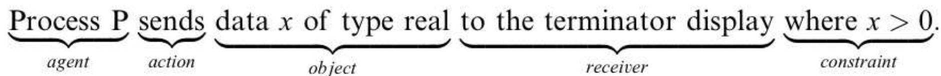

Figure 5.6 : Structure d'une étape selon [Zhu *et al.* 2002]

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 depuis 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 permettra 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].

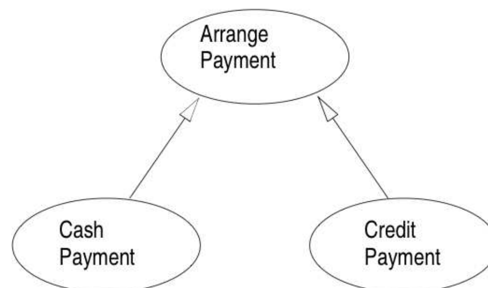

Figure 5.7 : Exemple de la généralisation entre use cases [Stevens 2007]

5.3.2 Sémantique de la relation d'inclusion entre les XUC

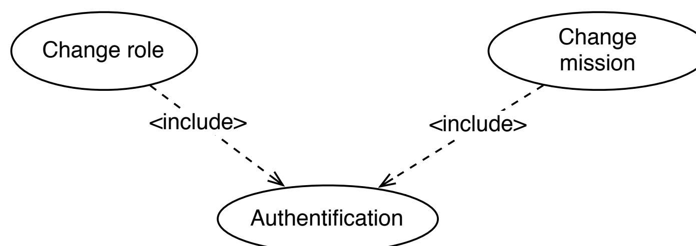

Figure 5.8 : Exemple de la relation include

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 cohérence 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.