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. Ensuite, 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

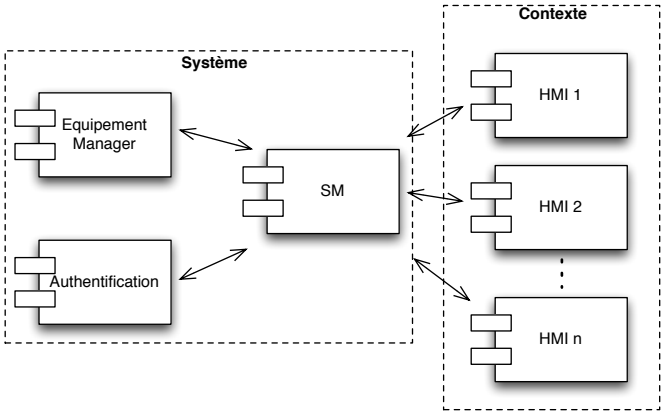

Figure 3.1 : Le composant SM au sein de son environnement

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.

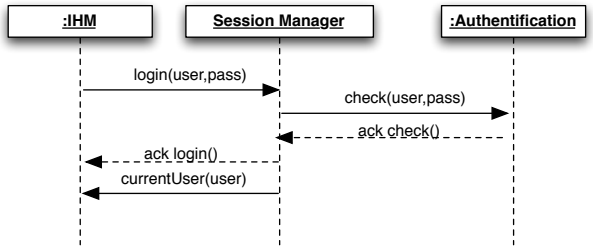

Figure 3.2 : Exemple de scénarios d'interaction du SM.

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 loging. 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-

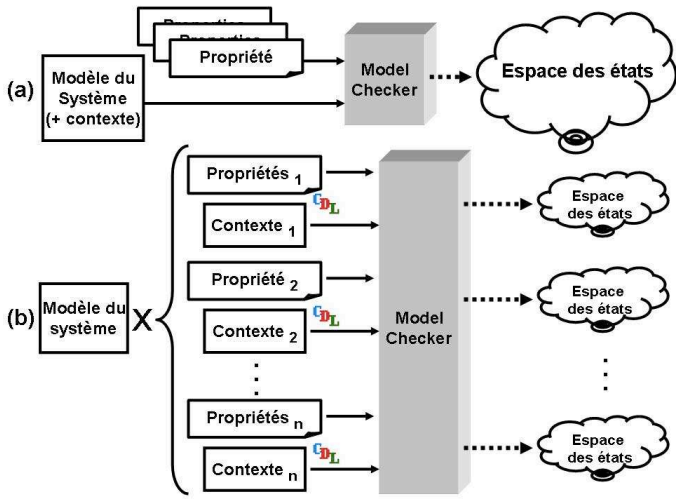

Figure 3.3 : Exploration sans (a) et avec (b) identification séparée des contextes.

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:

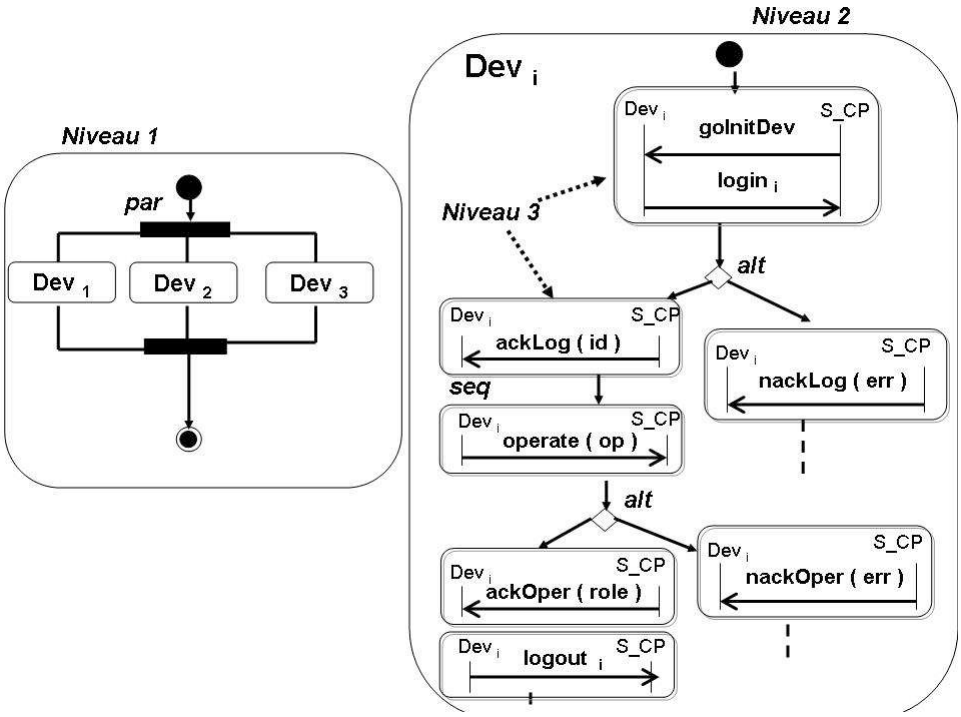

Figure 3.4 : Illustration d'un modèle CDL partiel du contexte du système SM

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

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.

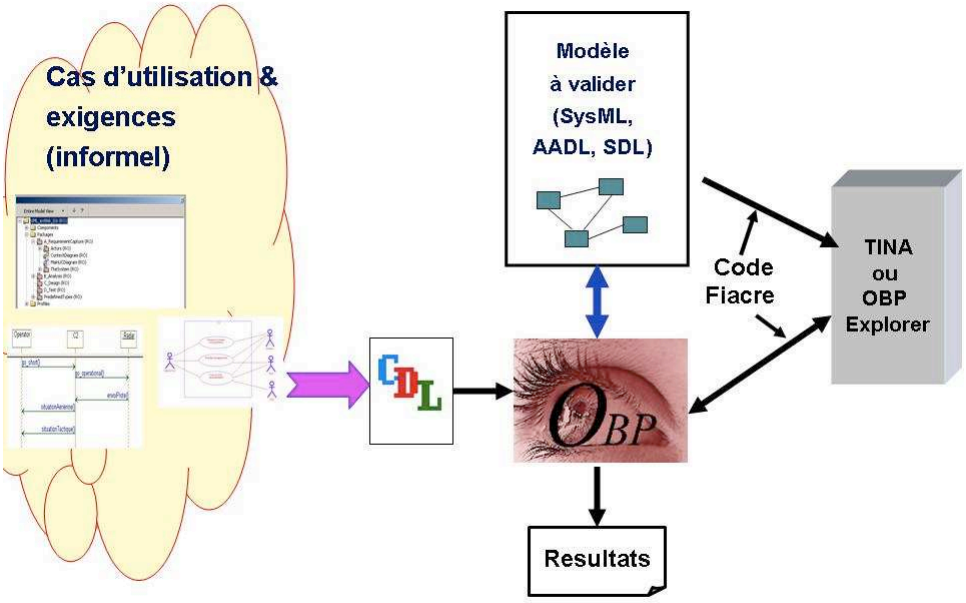

Figure 3.5 : Transformation de modèle CDL avec OBP

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].

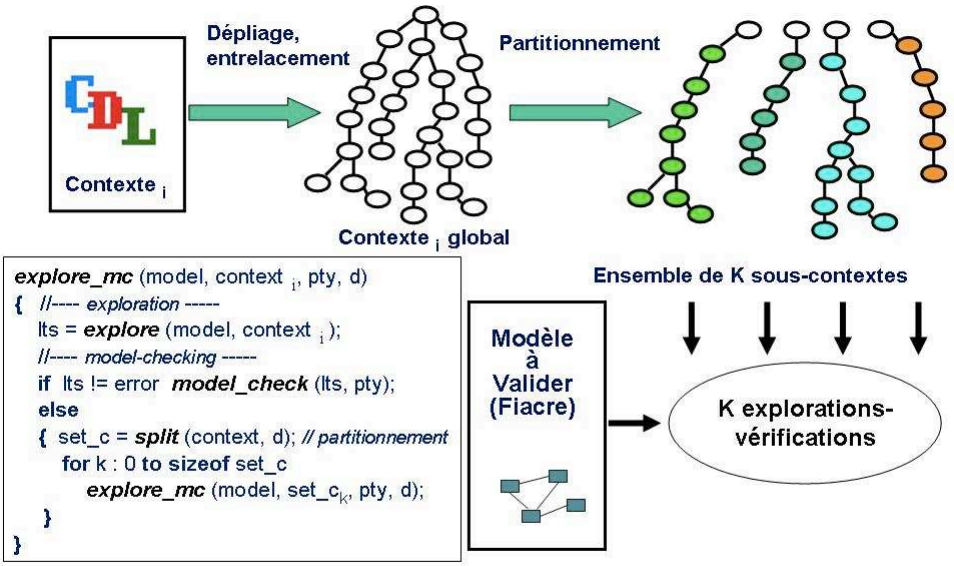

Figure 3.6 : Partitionnement d'un contexte et vérification pour chaque partition

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.

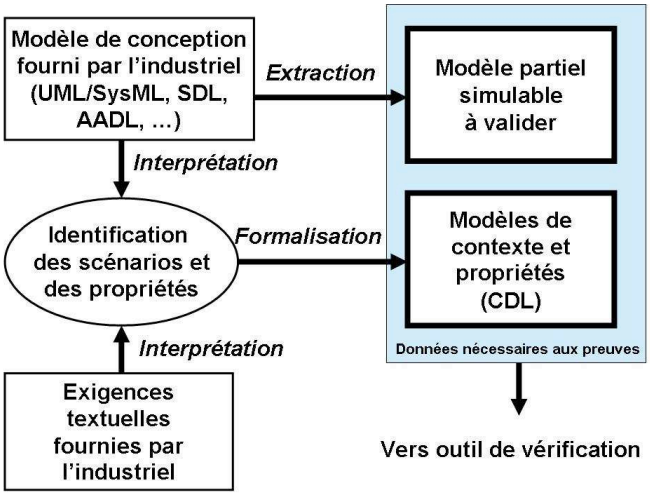

Figure 3.7 : Processus de prise en compte des données d'exigences et du modèle à valider.

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 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".