Un langage de contexte de preuve pour la validation formelle de modèles logiciels

Philippe Dhaussy[*], Julien Auvray*
Stéphane De Belloy**, Frédéric Boniol***, Eric Landel[* +]

* Laboratoire DTN, ENSIETA, BREST, F-29806 cedex 9
{dhaussy, auvrayju, landeler} @ensieta.fr
http://www.ensieta.fr/dtn
** THALES AIR SYSTEMS, BP 20351 94628 RUNGIS Cedex
stephane.debelloy@fr.thalesgroup.com
*** IRIT-ENSEEIHT, 2 rue C. Camichel BP 7122 – F-31071 Toulouse, cedex 7
frederic.boniol@enseeiht.fr
+ CS-SI, 6, avenue Saint Granier, Toulouse

Résumé. Pour améliorer les pratiques dans le domaine de la validation formelle de modèles, nous explorons un axe de recherche dans lequel nous formalisons la notion de « contexte de preuve » intégrant la description du comportement de l'environnement interagissant avec le modèle et les propriétés à vérifier dans ce contexte. L'article présente le langage CDL (*Context Description Language*) proposé à l'utilisateur pour la description des contextes de preuve. Ceux-ci sont exploités, actuellement dans nos travaux, par une technique de vérification de type *model-checking* avec la mise en œuvre d'observateurs. Dans une approche Ingénierie Dirigée par les Modèles (IDM), les modèles de contextes sont transformés en modèles d'automates temporisés puis en codes exploitables par l'outil OBP/IFx (*Observer-Based Prover*). Ce travail a donné lieu à plusieurs expérimentations industrielles comme la validation formelle d'un protocole de communication avionique pour l'AIRBUS A380. Dans cet article, nous décrivons l'application de notre approche pour la validation d'un modèle de contrôleur de système aérien conçu par THALES. L'article rend compte de la mise en œuvre du langage CDL et d'un retour d'expérience.

1 Introduction

La validation formelle des architectures logicielles. Dans le domaine des systèmes embarqués, les architectures logicielles doivent être conçues pour assurer au sein de ceux-ci des fonctions de plus en plus vitales. Les architectures de calculateurs comme ceux des domaines avioniques ou automobiles, les systèmes d'informations critiques ou d'acquisition de données sont soumis à des contraintes de temps et de fiabilité très importantes. Leur développement nécessite donc des techniques d'ingénierie prenant en compte ces caractéristiques dès les phases amont de leur cycle de vie. Par exemple, en ce qui concerne la gestion des exigences, spécifiées suite à l'expression du besoin des utilisateurs, et exploitées

dans le développement d'un système, les activités de validation et de vérification sont d'une importance cruciale car elles impactent très souvent les phases en aval du cycle. En effet, une mauvaise spécification ou compréhension des exigences client et la mauvaise mise en œuvre dans les choix de conception des architectures logicielles peuvent amener à des dysfonctionnements remettant fortement en cause la fiabilité et la sûreté des systèmes (Standish group, 1995).

Dans les architectures logicielles des systèmes d'information embarqués, une large partie des constituants sont des composants réactifs, de nature événementielle, qui organisent l'enchaînement des actions en fonction des interactions, soit inter-composants, soit avec l'environnement du système, et des événements reçus. A titre d'exemple, le développement des applications de contrôle du trafic aérien, en particulier les ATC (*Air Trafic Control*) développées pour les flottes d'AIRBUS, mettent en œuvre de nombreux composants interagissant fortement. Ces sous-systèmes étant parallèles et asynchrones, il est important d'ordonnancer leurs actions et assurer un contrôle fiable de l'ensemble du système. Lors de la conception des sous-systèmes, il est notamment primordial de démontrer l'absence de blocage et la possibilité pour chacun d'entre eux d'assurer un fonctionnement correct et dans des délais compatibles avec leurs propres exigences temporelles. Il est également important de démontrer que, dans les divers modes de fonctionnement prévus, nominaux ou dégradés, les comportements des composants en interaction correspondent aux exigences exprimées compte tenus des différents scénarios d'utilisation durant les phases opérationnelles.

Pour répondre à ce défi, les méthodes formelles et en particulier les méthodes de vérification de comportements sont explorées depuis plusieurs années par de nombreuses équipes de recherche (Hooman *et al.*, 2007), mais aussi par des systémiers comme Airbus, Dassault Aviation ou Thales. Néanmoins, nous constatons aujourd'hui encore leur trop faible pénétration dans le processus d'ingénierie système et logicielle comparativement aux énormes besoins en termes de recherche de fiabilité et sûreté de fonctionnement des systèmes critiques. Cette contradiction trouve en partie ses causes dans la difficulté réelle de manipuler des concepts théoriques et des méthodes formelles dans un cadre industriel. De nombreux problèmes sont encore non résolus quant aux traitements de systèmes complexes et soumis à des contraintes fortes (temps réel, criticité, déploiement).

Des contextes pour restreindre les comportements du modèle. Dans le cas des systèmes réels, les modèles sont souvent de grande taille et leur vérification avec des méthodes classiques (*model-checking* par exemple) est limitée par le problème bien connu de l'explosion combinatoire (explosion du nombre d'états caractérisant le comportement du modèle). Pour contourner ce problème, beaucoup de travaux ont été entrepris pour identifier des techniques de réduction et d'abstraction de modèles en vue de leur validation. Parmi celles-ci, certaines reposent sur un encodage du contexte dans les propriétés à vérifier. Ceci pose le problème difficile de la formulation d'un contexte à l'aide de formules logiques, en particulier dans le cas d'environnements asynchrones composés de multiples agents.

D'autres techniques utilisent des automates de contexte, spécifiques et intrusifs, qui simulent des scénarios de comportement de l'environnement en vue de limiter l'espace d'exécution du modèle lors de la preuve. Ces automates de contexte sont composés avec le modèle à valider ce qui permet de restreindre l'ensemble des exécutions du modèle. L'intérêt

méthodologique de cette méthode est de permettre d'initialiser le système dans des configurations qui intéressent l'utilisateur en identifiant les séquences « désirées » ou « non désirées ». L'intérêt pratique en est également d'être utilisable sur des modèles de grande taille, en recherchant à réduire l'espace des états du modèle par la composition du modèle du système à vérifier avec ces automates de contexte et de restriction.

Dans le cas d'applications plongées dans un environnement faisant intervenir de nombreuses entités, la description du contexte environnemental peut ici encore ne pas être une tâche aisée. Il faut donc que l'utilisateur puisse disposer d'un langage et d'un outil lui permettant de le décrire de manière simple. L'activité de preuve d'exigences sur un modèle logiciel implique aussi d'une part une formalisation de ces exigences, et d'autre part de préciser l'environnement, c'est-à-dire le contexte, qui interagit avec le modèle soumis à validation. D'un point de vue notation ainsi que méthodologique, la plupart des travaux sur la vérification formelle n'offre pas la notion de contexte et impose à l'utilisateur de simuler implicitement l'environnement comme un sous-système à l'intérieur d'un système globalement fermé (aucune entrée ni sortie). Très souvent, ce contexte est décrit, de manière textuelle, dans des documents élaborés durant les phases amont du cycle de vie, lors des activités d'analyse. Aussi, ces descriptions font peu le lien entre le comportement de l'environnement et les propriétés à vérifier.

Il nous apparaît donc indispensable d'étudier un cadre structurant pour décrire et formaliser des *contextes de preuve* (Dhaussy *et al.*, 2007), en tant que composant MDA (Clarke *et al.*, 2004) décrivant conjointement les exigences à vérifier et les différents comportements de l'environnement dans lesquels le modèle est plongé lors des simulations et des analyses formelles. Ainsi, nous pensons que cette approche peut contribuer à une meilleure intégration des techniques de vérification dans le processus d'ingénierie (Blanc *et al.*, 2006). Dans ce but, nous avons donc défini un langage nommé CDL (*Context Description Language*) (De Belloy *et al.* 2007). Il constitue le cœur de cet article et a été expérimenté sur deux cas industriels.

Organisation de l'article. Nous décrivons le langage CDL proposé en section 2 ainsi que son exploitation dans l'outil prototype OBP. Une des deux expérimentations menées est introduite en section 3 avec la mise en œuvre d'une description CDL. Nous donnons quelques résultats de ces travaux en section 4 et en tirons un premier bilan. Nous concluons et présentons nos perspectives en section 5.

2 Le langage CDL pour l'expression des contextes et des exigences

2.1 Le langage de définition des contextes

Un modèle CDL permet à l'utilisateur de décrire le comportement de l'environnement du modèle à valider et les propriétés devant être vérifiées. Nous avions proposé précédemment (Roger, 2006, Dhaussy *et al.*, 2007) un langage de contexte exploitant des diagrammes de type UML2, inspirés des *Use Case Charts (UCC)* de (Whittle, 2005). Dans cette première version, ce langage permettait de décrire l'environnement en tant qu'entité unique et

interagissant séquentiellement avec le modèle. Nous étendons ce langage pour permettre, d'une part, de décrire plusieurs entités contribuant à l'environnement et pouvant s'exécuter en parallèle. D'autre part, nous intégrons un langage de description de propriétés reposant sur la notion de patron que nous décrivons succinctement au paragraphe 2.2. Un méta modèle de CDL a été défini et une sémantique décrite (De Belloy *et al.*, 2007) en terme de traces, s'inspirant des travaux de (Hauguen, 2005, Roger, 2006).

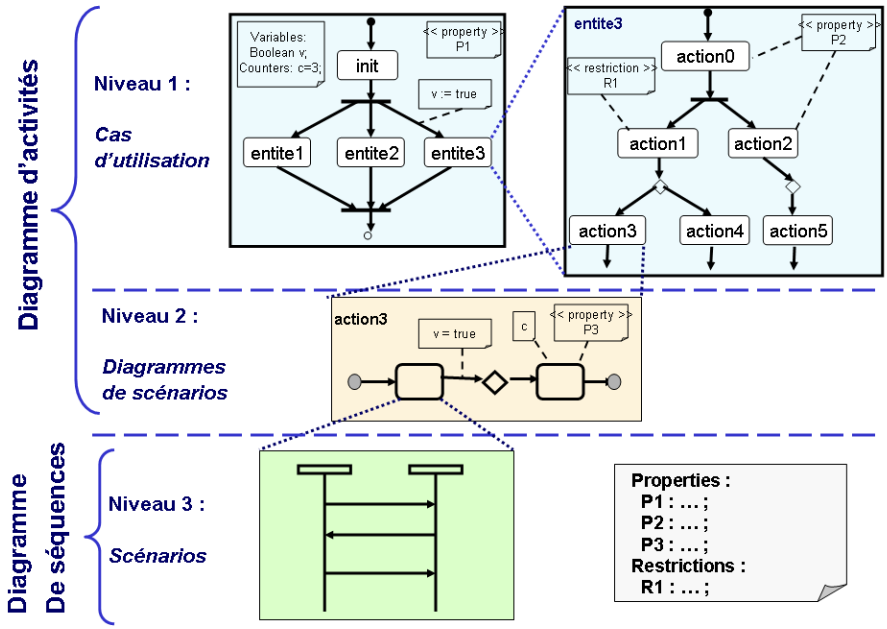

Figure 1. *Les trois niveaux d'un modèle CDL (Context Description Language)*

Un modèle *CDL* est structuré, de manière hiérarchique, en 3 niveaux (figure 1[1]). Au premier niveau, des diagrammes de cas d'utilisation décrivent, par des diagrammes d'activités et de manière hiérarchique, des enchaînements d'activités des entités s'exécutant en parallèle et constituant l'environnement. Les diagrammes de ce niveau font référence à des diagrammes de scénarios décrits au niveau 2 également sous la forme de diagrammes d'activités. Ces diagrammes décrivent des enchainements de scénarios, ceux-ci étant décrits au niveau 3 par des diagrammes de séquence UML2.0 simplifiés. Le langage est conçu dans le but d'offrir à l'utilisateur un cadre simple pour spécifier les enchainements de scénarios qui décrivent les interactions entre le modèle soumis à validation et des entités composant l'environnement de ce modèle. 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 souhaitable que l'utilisateur spécifie un ensemble de modèles CDL, chacun correspondant à des cas d'utilisation du modèle à valider.

[1] La figure 1 est donnée ici en guise d'illustration des concepts intégrant le méta modèle CDL. La syntaxe concrète du langage n'est pas encore définitive et n'est donc pas donnée dans cet article.

Description du niveau 3. Les diagrammes de niveau 3 sont des scénarios. Chaque diagramme décrit une séquence d'interactions entre un acteur de l'environnement et le modèle à valider sous forme d'un diagramme de séquences UML2.0 simplifié. La sémantique d'un scénario est exprimée par un ensemble de traces comme décrit dans (Hauguen, 2005) et conformément à la sémantique des diagrammes de séquence d'UML2.0 (OMG, 2003). Dans une interaction simple, c'est-à-dire impliquant un seul message x, l'envoi d'un message (*x!*) précède la réception de ce message (*x?*). Une trace d'un scénario *S* est une séquence ordonnée d'évènements et décrit un historique de l'interaction entre objets, comme par exemple la trace < *x!*, *x?* > pour l'interaction simple précédente. Un scénario, impliquant plusieurs interactions, est associé à un ensemble de traces qui sont construites à l'aide des opérateurs d'interaction *seq, alt* et *par* d'UML2.0.

Description du niveau 2. Au second niveau, Les diagrammes de scénarios, sont des diagrammes d'activités UML2 simplifiés dont certains nœuds référencent des scénarios décrits au niveau 3. Un nœud de ces diagrammes peut être de différents types : Soit un nœud qui référence un scénario de niveau 3, soit un nœud de séquence ou de choix permettant de composer des scénarios, soit un nœud initial ou un nœud final. Chaque transition peut être gardée par une expression booléenne référençant des variables. Les diagrammes de scénarios acceptent 3 types de nœuds finaux. Le nœud *ok* (normal) indique la terminaison de l'exécution du diagramme. Le nœud *cancel* permet de relancer l'exécution au niveau du nœud appelant au niveau 1. Le nœud *stop* arrête l'exécution en cours. La sémantique d'un diagramme de scénarios s'appuie sur la sémantique des scénarios et est exprimée par des règles de construction d'ensembles de traces à l'aide des opérateurs *seq* et *alt*.

Description du niveau 1. Les diagrammes de niveau 1 sont des diagrammes d'activité UML2.0. Ils décrivent, de manière hiérarchique, des enchaînements d'activités des entités constituant l'environnement. Pour chaque diagramme, les nœuds sont de différents types. Un nœud peut référencer un diagramme de niveau 2 ou un sous-diagramme de niveau 1. Un nœud peut également être un opérateur indiquant une alternative entre plusieurs exécutions, une mise en parallèle de plusieurs exécutions ou indiquer une sortie du diagramme. La sémantique d'un diagramme de niveau 1 s'appuie également sur la sémantique des scénarios et est aussi exprimée par des ensembles de traces construits à l'aide des opérateurs *seq, alt* et *par*.

Dans les diagrammes de niveau 1 et 2, des ensembles de compteurs, de variables et de sémaphores sont identifiés. L'objectif des compteurs est de permettre de limiter les boucles d'exécution des entités de l'environnement. Chacun d'entre eux est associé à un nœud d'un diagramme. La gestion des compteurs permet d'assurer, comme décrit dans (Roger, 2006), un dépliage fini lors de la construction des automates du contexte dans le langage d'implantation de l'outil de preuve choisi. Les variables permettent quant à elles, de mémoriser des états de l'environnement. Elles peuvent être référencées dans des gardes conditionnant le séquencement dans les diagrammes d'activités. Les sémaphores sont utilisés pour spécifier des opérations de synchronisation entre acteurs.

En plus de ces trois niveaux, un modèle CDL intègre la spécification des propriétés à vérifier et des restrictions. Les propriétés spécifiées sont référencées dans le modèle par des liens stéréotypés *property*. Une propriété peut être liée à un des diagrammes des deux

premiers niveaux. Dans ce cas (exemple de la propriété P1 de la figure 1), la propriété sera vérifiée pour les exécutions référencées dans ce diagramme. Une propriété peut être aussi liée à un nœud dans un diagramme de niveau 1 ou 2 (exemple des propriétés P2 et P3). Elle sera à vérifier pour les exécutions associées à ce nœud. L'intérêt est de lier une propriété à un contexte d'exécution et de préciser explicitement les conditions d'application de l'exigence, ce qui est rarement fait dans le cas de documents d'exigences industriels que nous avons eu à traiter. Dans les documents d'exigences d'un système, celles-ci sont souvent exprimées dans un contexte donné de l'exécution du système. Ce contexte correspond à des phases opérationnelles bien définies, comme par exemple les phases d'initialisation, de reconfiguration, de modes dégradés, de changement d'état, etc. Pour une phase donnée, toutes les propriétés du cahier des charges ne sont pas à prendre en compte mais uniquement un sous ensemble de celles-ci. A la lecture des cahiers des charges, cette information contextuelle est très souvent mal explicitée, voire implicite ou disséminée dans plusieurs documents. Dans les travaux décrits dans **(**Dwyers *et al.*, 1999, Konrad *et al.*, 2005), les annotations du type scope permettent de localiser les exigences à vérifier dans un contexte temporel d'exécution du modèle à valider. Mais en pratique, dans des contextes d'exécution complexes, ces annotations ne sont pas aisées pour spécifier cette localisation. Le déroulement des phases d'exécution de l'environnement peut être difficile à décrire avec les opérateurs proposés à cause de l'enchaînement des interactions entre l'environnement et le modèle. Pour pouvoir rendre opérationnelle l'utilisation de patrons de définition, nous proposons donc d'associer chaque propriété à vérifier à un nœud d'un modèle de contexte CDL. L'intérêt est d'associer les propriétés à une phase d'exécution de l'environnement. Les propriétés sont localisées dans un contexte temporel déterminé par le cahier des charges. Dans une approche de vérification par observateurs, l'observateur encodant une propriété est pris en compte (*enabled*) uniquement dans le nœud d'exécution du contexte et désactivé (*disabled*) en dehors du nœud. Le mécanisme de prise en compte (*enabled/disabled*) des propriétés a pour conséquence de réduire l'explosion lors de la composition du contexte, des observateurs et du modèle. Lors de la génération du graphe des exécutions du système complet, des chemins sont supprimés grâce à la prise en compte des états *enabled-disabled* des observateurs. Certaines propriétés pourront, quant à elles, être prises en compte durant toute l'exécution de l'environnement. Dans ce cas, les observateurs associés ont un statut particulier et sont référencés globalement dans le contexte CDL.

Les restrictions permettent d'interdire volontairement certaines exécutions du modèle. Les automates de restriction précisent les conditions de coupure du graphe d'exécution généré par la composition. L'intérêt des restrictions est de préciser les exécutions qui n'intéressent pas l'utilisateur et de permettre de réduire le graphe des exécutions observées en simulation. Dans l'état actuel du langage CDL et de son exploitation dans l'outil prototype, les automates de restriction ne sont pas encore inclus dans le langage, mais sont exprimés sous la forme d'automates IF2. A terme, les restrictions seront associées (liens stéréotypés *restriction*), comme pour les propriétés, à un diagramme ou un nœud des niveaux 1 et 2. La restriction s'applique donc dans le contexte auquel elle est associée.

Dans le processus de modélisation du système, les diagrammes CDL, la définition pertinente des automates de restrictions et la valeur des compteurs doivent être fournies comme un résultat du processus d'analyse. Comme décrit dans (Dhaussy *et al.*, 2007), la construction des modèles CDL inclus dans un ensemble d'unités de preuves doit être

accompagnée d'une méthodologie permettant d'aider l'utilisateur à définir tous les contextes environnementaux qui doivent être pris en compte.

2.2 Le langage de définition des propriétés

Une difficulté de mise en œuvre d'une technique par *model-checking* est de pouvoir exprimer les propriétés à vérifier de façon aisée. Les langages à base de logique temporelle permettent en théorie une grande expressivité des propriétés. Mais en pratique dans un contexte industriel et au regard de la grande majorité des documents d'exigences à manipuler, ces langages sont souvent difficiles, voire impossible à utiliser tel quel. En effet, une exigence peut référencer de nombreux événements, liés à l'exécution du modèle ou de l'environnement, et est dépendante d'un historique d'exécution à prendre en compte au moment de sa vérification. Ceci implique que les formules logiques à exprimer sont d'une grande complexité et deviennent difficilement lisibles ou manipulables par les ingénieurs. Il est donc nécessaire de faciliter l'expression des exigences avec des langages adéquats permettant d'encadrer l'expression des propriétés et d'abstraire certains détails, au prix de réduire l'expressivité. De nombreux auteurs ont fait ce constat depuis longtemps et certains (Dwyers *et al.*, 1999, Smith *et al.*, 2002, Konrad *et al.*, 2005) ont proposé de formuler les propriétés à l'aide de patrons de définition. Nous reprenons cette approche et nous l'implantons dans le langage de description de contextes CDL.

Les patrons capturent, sous forme textuelle, des types de propriétés usuellement rencontrées dans les documents d'exigences. Chez les auteurs précédents, les patrons sont classés en familles de base et prennent en compte les aspects temporisés des propriétés à spécifier. Les patrons identifiés dans une première approche permettent d'exprimer des propriétés de réponse **(***Response*), de pré-requis (*Precedence*), d'absence (*Absence*), d'existence (*Existence*), d'universalité (*Universality*). Les propriétés font référence à des évènements détectables comme des envois ou des réceptions de signaux, des actions, des changements d'état. Les formes de base peuvent être enrichies par des options (*Pre-arity, Post-arity, Immediacy, Precedency, Nullity, Repeatability*) à l'aide d'annotations (Konrad *et al.*, 2005). Les auteurs ont proposé d'identifier la portée (*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).* Le *scope* indique si la propriété doit être prise en compte, par exemple, durant toute l'exécution du modèle, avant, après ou entre des occurrences d'évènements. Nous enrichissons les patrons avec la possibilité d'exprimer des gardes sur les occurrences d'évènements exprimées dans les propriétés. En effet, il est souvent utile de pouvoir permettre ou non la prise en compte de la détection d'un évènement en fonction de l'état de l'environnement. Une occurrence d'évènements exprimée dans une propriété peut donc être associée à une garde référençant des variables déclarées dans le modèle CDL. Une autre extension apportée aux patrons est la possibilité de manipuler des ensembles d'évènements, ordonnés ou non ordonnés comme dans la proposition de (Janssen *et al.*, 1999). Les opérateurs *an* et *all* précisent respectivement si un évènement ou tous les événements, ordonnés (*Order*) ou non (*Combined*), d'un ensemble sont concernés par la propriété. Nous donnons informellement, en figure 2, la forme générale d'un patron de type réponse et, en figure 5, un exemple de propriété de vivacité bornée exprimée avec ce type de patron, sur le cas d'étude décrit.

Property *name* :
Scope [
[**An** | **All**] [**Order** | **Combined**]
[**if** (*condition_m0*)]
[*Pre-arity*] **occurrences of** *x0_0*
…
[*Pre-arity*] **occurrences of** *x0_k*
…
[**if** (*condition_ml*)]
[*Pre-arity*] **occurrences of** *xl_0*
…
[*Pre-arity*] **occurrences of** *xl_k'*
End_order
[*Immediacy*] **leads-to** [< to]
[**An** | **All**] [**Order** | **Combined**]
if (*condition_n0*)
[*Post-arity*] **occurrences of** *y0_0*
…
if (condition_nl)
[Post-arity] **occurrences of** *yl_k'*
End_order
x0_0 [*Nullity*] **occurs**, …, *xl_k'* [*Nullity*] **occurs**
One of [*y0_0, …, yl_k'*] [*Precedency*] **occurs before the first one of** [*x0_0, …, xl_k']*
[*Repeatability*]]

Figure 2 : *Forme générale d'un patron de type réponse.*

Les options *Pre-arity* et *Post-arity* peuvent prendre les valeurs *Exactly one* ou *One or more*. L'option *Immediacy* prend les valeurs *Immediately* ou *Eventually* ce qui permet de préciser la prise en compte ou non d'occurrence d'évènements particuliers. Les options *Precedency*, *Nullity* et *Repeatability* prennent respectivement les valeurs *Cannot* ou *May*, *May Never* ou *Must et Repeatable* ou *Not repeatable* (voir les détails dans De Belloy *et al*., 2007).

2.3 Exploitation des modèles CDL dans l'outil prototype OBP[2]

Pour mener nos expérimentations, nous nous sommes basés sur le langage d'implantation formel IF2.0. Celui-ci repose sur le formalisme étendu des automates temporisés (Alur *et al*., 1994) adapté au contexte des systèmes communicants de manière asynchrone. La technique de vérification est basée sur la manipulation d'automates observateurs (Halbwachs *et al*., 1993). Le laboratoire VERIMAG a développé l'outillage *IFx* (Bozga *et al*., 2002, Ober *et al*., 2003) permettant la simulation de programmes IF2.0 et la génération des graphes d'accessibilité. Autour de cet outil, nous avons développé un outil nommé *OBP* qui intègre

[2] OBP vers.1.0, développé au laboratoire DTN (ENSIETA), est disponible sous licence EPL à l'adresse : http://gforge.enseeiht.fr/projects/obp. Dans cette version, OBP prend les contextes en entrée directement sous la forme d'*α-contexte*s au format XML.

IFx. Dans la configuration actuellement développée (version 2.0), les modèles CDL sont édités dans un environnement Eclipse et importés dans OBP au format XMI[3]. OBP les interprète et génère des automates dans le format intermédiaire des *α-contexte*s proposé dans (Roger, 2006) que nous avons étendu pour prendre en compte les exécutions parallèles des entités de l'environnement, la gestion des variables et des sémaphores. Ces automates sont ensuite composés, dépliés et partitionnés (Roger, 2006) pour produire des automates temporisés linéaires IF2.0 qui représentent l'ensemble des chemins d'exécution de l'environnement. Ce sont ces chemins qui sont composés avec les observateurs et le modèle à valider. C'est cette partition du contexte en un ensemble de chemins d'exécution qui permet d'aboutir, lors de la composition, à des graphes d'états de taille limitée rendant possible l'analyse d'accessibilité.

L'outil génère (figure 3), par une technique de transformation de modèles, les automates observateurs et de restriction à partir des propriétés et des restrictions qui sont décrites dans le modèle CDL. Dans le cas des observateurs, les propriétés que nous savons manipuler sont de type sûreté et vivacité bornée pour lesquelles nous pouvons concevoir des algorithmes de traduction en automates à partir des propriétés exprimées à l'aide des patrons. Des règles de transformation, mises en œuvre dans le langage KerMeta (Muller *et al., 2005*) permettent l'import des diagrammes CDL et leur prise en compte au sein d'OBP. Les modèles à valider dans OBP sont importés, quant à eux, actuellement au format IF2.0.

Dans notre prototype, les modèles CDL sont traduits en *α-contextes* par l'exécution de programmes de transformations de modèles spécifiés en langage KerMeta. Un *α-contexte* est un automate dont les étiquettes de transitions sont associées à des actions composées ou *α-actions* et référencent les compteurs présents dans les diagrammes CDL. Les actions sont composées d'un ensemble d'actions élémentaires. Les *α-contextes* sont destinés à être raffinés ou *dépliés* en des automates temporisés. La traduction des *α-contexte*s implique plusieurs phases automatisées et implantées par des programmes java :

- le dépliage des *α-contextes* en des automates nommés *contextes concrets* : cette phase consiste à remplacer, pour chaque transition de l'*α-contexte*, l'*α-action* par l'ensemble des actions élémentaires. Le dépliage est contrôlé en prenant en compte l'ensemble des valeurs des compteurs, ce qui assure la terminaison de la génération du contexte concret.

– la génération d'un automate résultant de l'entrelacement des *contextes concrets* avec la prise en compte de leurs exécutions parallèles.

– le partitionnement de l'automate en un ensemble d'automates de *chemins* : cette phase consiste à produire des automates acycliques permettant de découper le contexte en des chemins d'exécution finis et traduisibles en langage IF2.0.

– Enfin, chaque chemin généré est transformé en un automate IF2.0 et composé avec le système et les automates observateurs et les automates de restrictions.

[3] XML (*eXtensible Markup Language*) *Metadata Interchange*

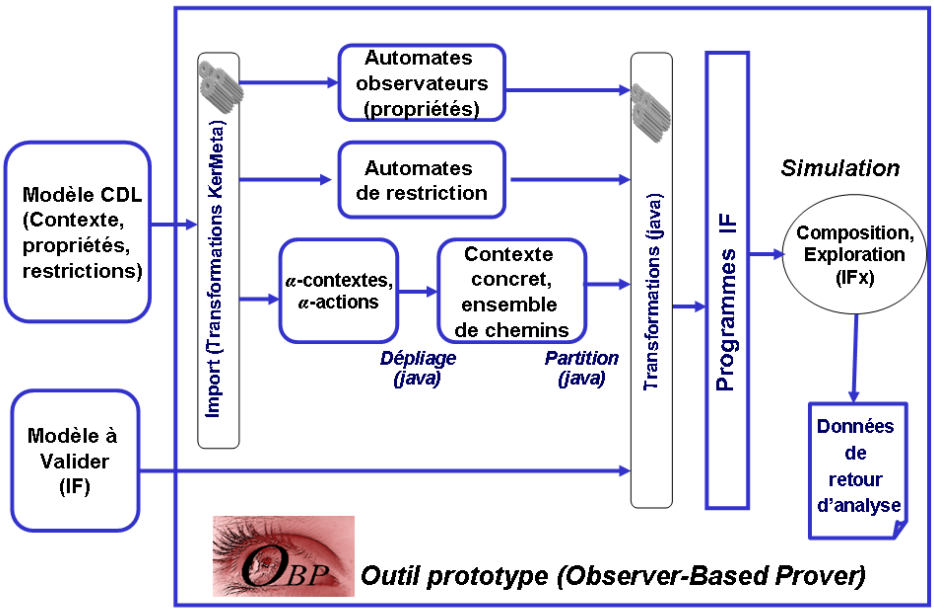

Figure 3. *L'outil prototype OBP/IFx (vers 2.0)*

Pour prendre en compte les imports de modèles au format type UML2, AADL (SAE, 2004) ou SDL, il est encore nécessaire de mettre en œuvre des traducteurs adéquats en s'inspirant des résultats des travaux menés dans des projets comme par exemple TopCased[4] ou Omega[5]. L'analyse d'accessibilité est réalisée sur le résultat de la composition entre un chemin, un ensemble d'observateurs et le modèle à valider. S'il existe un état atteint *reject* d'un observateur de propriété pour l'un des chemins, alors la propriété est considérée comme n'étant pas vérifiée.

3 Mise en œuvre de CDL sur des cas d'étude industriels

L'outil OBP version 2.0, en cours de développement, et le langage CDL ont été expérimentés pour la validation d'architectures logicielles sur deux études de cas industrielles, la première sur le système ATC (*Air Traffic Control*) de l'A380 en collaboration avec Airbus et CS-SI, le second sur un système militaire anti-aérien avec Thales. La première étude de cas est décrite dans (Dhaussy *et al*, 2007). Nous nous concentrons dans cet article sur la seconde étude de cas.

[4] http://www.topcased.org
[5] http://www-omega.imag.fr

3.1 Présentation du cas d'étude

Le cas industriel étudié chez Thales est composé de la partie logicielle, nommée S_CP[6], du système de « *management* » d'un système anti-aérien. Le rôle de S_CP est de commander et contrôler l'ensemble du système, ses modes et ses actions en réponse aux informations ou observations reçues de l'environnement. Il s'agit d'un système critique soumis à des exigences de fiabilité et de sûreté. Ses fonctionnalités sont plus précisément :

- Contrôler l'état global du système (*Shutdown, Standby, Short_Notice, Minimal_Notice, Immediate_Notice*). Cet état représente le degré d'activation du système. Selon les commandes de l'opérateur, S_CP exécute et surveille le bon fonctionnement des opérations de changement de modes (vérification des conditions de changement de mode, exécution des actions de clôture et d'initialisation de mode…).
- Contrôler et déterminer la configuration globale du système (*Revised_Autonomous, Integrated*) indiquant l'acteur autorisé à interagir avec le système.
- Percevoir et gérer l'état de l'environnement du système (*Normal, In_Fault, In_Assessment, Degraded*) correspondant aux états de fonctionnement des différents équipements environnant le système.

Ce système a été modélisé en UML Rhapsody et implémenté par un programme ADA d'environ 38000 lignes. Le modèle comporte un automate de 365 états et 560 transitions. A des fins de validation, ce modèle a été traduit sous la forme d'un automate formel IF2.0. Le nombre d'exigences exprimées dans le cahier des charges du système est de 170.

3.2 Descriptions CDL

La première étape dans la validation de ce système a consisté en la formalisation à la fois de son environnement et de ses propriétés à l'aide du langage CDL. A titre d'exemple nous donnons, figure 4, un extrait du modèle CDL, modélisant les interactions entre le système S_CP et un acteur de son environnement. Nous ne montrons que les niveaux 1 et 3 du modèle par souci de simplification.

Ce CDL est composé d'un diagramme de mode de premier niveau, décrivant l'enchaînement de deux modes séquentiels suivis, en cas de terminaison correcte de ces modes, des trois modes exécutés en parallèle (*Gérer_état*, *Gérer_configuration*, et *Gérer_condition*). Chaque mode de ce diagramme de premier niveau est ensuite raffiné en un diagramme de modes de second niveau (non donnés ici). Enfin, à chaque diagramme de deuxième niveau est associé un ou plusieurs diagrammes de séquence spécifiant les interactions possibles entre le système et son environnement. La spécification du contexte du système a nécessité la formalisation de 15 diagrammes de premier ou second niveau et une trentaine de diagrammes de séquence. Notons que l'objectif poursuivi ainsi n'était de pas formaliser entièrement et exhaustivement l'environnement du système et l'ensemble de leurs interactions, mais de spécifier les contextes de preuves nécessaires à la vérification des exigences du système. Les contextes décrits dans les diagrammes CDL, notamment au moyen des diagrammes de séquence, sont les contextes référençant soit les interactions directement adressées par les exigences à valider, soit les interactions influant indirectement par le comportement du système sur les réactions du système adressées par ces exigences.

[6] Le nom réel du système n'est pas donné dans cet article pour raison de confidentialité.

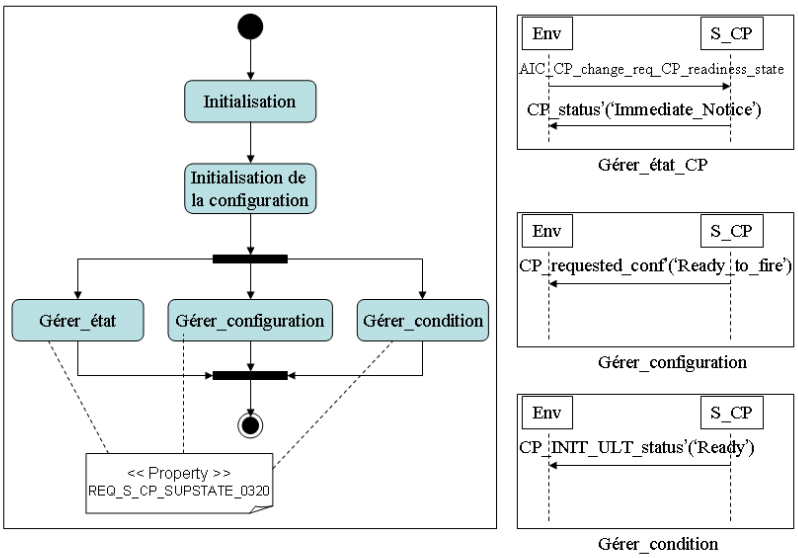

Figure 4 : *CDL (partiel) pour la description des interactions du système S_CP*

Les exigences à vérifier sont exprimées à l'aide de patrons permettant de formaliser principalement des propriétés de sûreté (invariance), de réponse bornée (p implique q avant un certain délai), ou d'accessibilité. A titre d'exemple nous donnons, figure 5, l'exigence REQ_S_CP_SUPSTATE_0320.

```
// commentaire …
Property REQ_S_CP_SUPSTATE_0320 :
Global [
  [exactly one] occurrences of AIC_CP_change_req_CP_readiness_state ;
  [eventually] leads_to [ <= 10 ]
All Order
  [exactly one] occurrences of CP_status('Immediate_Notice') ;
  [exactly one] occurrences of CP_requested_conf ('Ready_to_fire') or
                CP_requested_conf ('Not_Ready_to_fire') ;
  [exactly one] occurrences of CP_INIT_ULT_status ('Ready') or
                CP_INIT_ULT_status ('Fault') ;
 End order
AIC_CP_change_req_CP_readiness_state [ may never ] occurs ;
[repeatable] ]
```

Figure 5 : *Exemple d'exigence du système CP.*

Il s'agit d'une propriété de réponse bornée (« Reception message » … « leads_to » …) imbriquée dans une propriété d'invariance (« *Global* »). Cette exigence exprime qu'à tout

moment (« *Global* »), une requête de changement d'état (événement « *AIC_CP_change_req_CP_readiness_state* ») produit par l'opérateur doit provoquer l'émission par le système des trois événements spécifiés et dans l'ordre mentionné, à savoir : la notification « *CP_status (Immediate_Notice*) » indiquant que la requête a été prise en compte, puis après configuration et diagnostic du système la notification de la configuration « *Ready_to_fire* » ou en cas de panne détectée la configuration inverse « *Not_Ready_to_fire* », puis enfin la notification de condition de tir (« *Ready* » ou *Fault* »). Cette réponse doit être produite avant 10 unités de temps (ici des millisecondes). Les événements émis en réponse par le système étant produits dans des modes différents, la vérification de cette exigence doit être faite sur l'ensemble des trois modes (*Gérer_état*, *Gérer_configuration*, et *Gérer_condition*). Afin d'être formellement vérifiée, cette exigence est traduite sous la forme d'un automate observateur contenant un nœud de rejet dont l'accessibilité signifie la falsification de la propriété par le modèle du système. Cet automate observateur est donné figure 6.

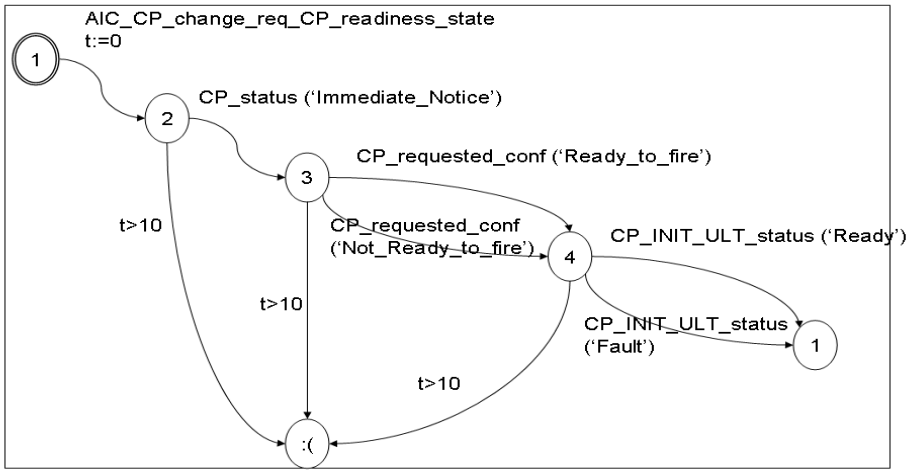

Figure 6 : *Automate observateur de la propriété REQ_S_CP_SUPSTATE_0320*

Le nœud de rejet sera atteint si et seulement si le timer *t* peut atteindre et dépasser la valeur 10 alors que l'un des états 2, 3 ou 4 est occupé. Inversement, la non accessibilité de l'état de rejet démontrera que, soit aucun message « *AIC_CP_change_req_CP_readiness_state* » ne peut être reçu (ce qui est contraire aux diagrammes CDL exprimés), soit la chaîne reliant le nœud 2 au nœud 1 (en passant par les nœuds 3 et 4) ne prend jamais plus de 10 millisecondes. La vérification de cette exigence sera effectuée sur le modèle du système composé avec le modèle de son contexte. Nous précisons que la traduction des propriétés en automates observateur n'est possible que pour les propriétés de type accessibilité, sûreté et de vivacité bornée, qui correspondent à une large majorité des propriétés à traiter.

4 Résultats et bilan

4.1 Appropriation du langage CDL

Dans les cas d'études traités, les modèles CDL ont été élaborés à partir, d'une part, des scénarios décrits dans les dossiers de conception et formalisés par des diagrammes de séquence et, d'autre part, à partir des documents répertoriant les exigences systèmes et dérivées. Lors de l'élaboration des modèles CDL, nous avons été confrontés à deux difficultés majeures. Une première est liée à un manque de formalisation complète et cohérente de la description du comportement de l'environnement des systèmes étudiés. Les scénarios d'utilisation des systèmes, et donc les interactions entre les modèles étudiés et leur environnement sont décrits dans plusieurs documents parfois de manière incomplète.

Certaines informations concernant des modes d'interactions étaient implicites. L'élaboration des modèles CDL a donc nécessité des discussions avec les experts maîtrisant les systèmes. La deuxième difficulté a concerné la compréhension des exigences. Celles-ci sont regroupées dans des documents de niveaux différents correspondants aux exigences systèmes ou dérivées. Ces dernières sont des exigences rédigées suite à l'interprétation des exigences systèmes au regard des choix de conception. L'ensemble des exigences analysées étaient rédigées sous forme textuelle et certaines d'entre elles donnaient lieu à plusieurs interprétations différentes. D'autres faisaient appel à une connaissance implicite du contexte dans lequel elles doivent être prises en compte. En effet, la plupart des exigences sont à prendre en compte dans une configuration donnée, lorsque le système a atteint une phase opérationnelle. Or les informations, concernant l'historique ayant mené à l'état dans lequel doit se trouver le système, ne sont en général pas explicités dans l'exigence. Elles font partie d'un ensemble d'informations qui doivent se trouver décrites explicitement dans les documents d'analyse du système. Mais parfois, elles ne le sont pas et doivent être alors déduites d'une interprétation sur le comportement du système, pouvant être faite par les experts, le connaissant parfaitement. Suite à notre proposition, au travers du langage CDL, de mise à disposition d'un cadre de formalisation des contextes et des exigences, les ingénieurs se le sont approprié et y ont trouvé immédiatement un intérêt pour mieux rédiger leurs spécifications. En ce sens, le couplage entre un modèle de type CDL et des ensembles de propriétés est une voie d'étude à poursuivre.

4.2 Preuves de propriétés établies

Dans chacun des deux cas d'étude traités, nous avons établi la preuve d'environ une trentaine d'exigences significatives de type vivacité bornée, pour les contextes que nous avons élaborés. Celles-ci ont été rédigées à l'aide d'un des patrons de propriétés proposés. Chaque propriété a été convertie, soit automatiquement quand l'algorithme de traduction était opérationnel, soit manuellement, en un automate observateur au format IF2.0. La conception des algorithmes de traduction se poursuit actuellement. Le pourcentage de propriétés, non traduisibles en un automate observateur, car ne rentrant pas dans la catégorie de sûreté ou d'accessibilité, est de moins de 20% de l'ensemble des exigences contenues dans les documents que nous avons étudiés. L'élaboration des contextes a permis de restreindre le comportement du modèle à valider par la génération des chemins, représentant

chacun un comportement de l'environnement. Chaque chemin, généré sous la forme d'un automate linéaire dont la taille en nombre d'états est comprise entre 100 et 300, a été composé avec l'observateur et le modèle. L'occurrence de la falsification d'une propriété dans chacun des modèles des deux d'études nous a permis de détecter une erreur résiduelle dans le modèle concerné. Le but de l'expérimentation était de démontrer la faisabilité de la technique pour un ensemble de contextes identifiés et d'exigences significatives. La poursuite de notre travail s'oriente aujourd'hui vers la méthodologie de construction des ensembles complets de contextes pour pouvoir assurer le bien fondé des preuves. La manipulation des contextes, dans notre approche, doit se faire avec précaution pour que les preuves des propriétés soient significatives dans la mesure où elles sont chacune réalisées pour des contextes restreints.

4.3 Intégration dans le processus de développement

Lors de l'étude des cas, une réflexion s'est engagée avec les ingénieurs de CS et Thales sur la capitalisation des activités de preuves menées. Notre proposition de structurer et mémoriser les données nécessaires à chaque preuve dans les unités de preuve décrites dans (Dhaussy *et al.*, 2007) a permis une meilleure structuration de l'activité. L'étude de l'intégration de ces composants MDA doit se poursuivre. Mais déjà la méthodologie suivie dans notre approche pour la description de chaque propriété a été suivie de la manière suivante : (1) choix d'une classe de patron exprimant l'intention de l'exigence, en prenant en compte les événements principaux devant être détectés ; (2) ajout des options et des gardes permettant de prendre en compte tous les événements pouvant intervenir lors des exécutions; (3) choix de la portée de la propriété dans un contexte (*Scope*) d'exécution à l'aide d'opérateurs temporels (*Global, Before, After, Between, After-Until, Repeatable, Not repeatable)* ; (4) enfin, liaison de la propriété au contexte opérationnel décrit en CDL. Les contextes quant à eux ont été élaborés d'une manière plus intuitive, à partir des documents d'analyses mis à disposition. Leur construction méthodique fait l'objet actuellement de travaux.

5 Conclusion et perspectives

L'objectif du travail présenté dans cet article est de contribuer à une meilleure intégration des techniques de vérification formelle dans un processus d'ingénierie industriel. Les expérimentations de vérification de propriétés menées sur les modèles logiciels, dans des contextes différents, et l'appropriation du langage CDL par les ingénieurs permettent d'entrevoir l'intérêt d'une telle approche. Les premiers retours d'expérience nous incitent donc à poursuivre ce travail et l'établissement d'une méthodologie. L'intérêt de la composition du modèle et des observateurs avec chaque chemin généré à partir de l'expression d'un contexte est de restreindre fortement le graphe d'exécution généré. Mais une limite de cette approche est liée à une explosion potentielle du nombre de chemins générés dans les cas où l'exécution de l'environnement est complexe. Un processus, encadré et outillé, de modélisation du système doit permettre de décrire et de partitionner en amont l'ensemble des scénarios du contexte.

Pour pouvoir capitaliser le savoir faire en matière de vérification, il nous semble important de structurer la démarche et les données manipulées lors des preuves. Pour cela, nous avons identifié (Dhaussy *et al*., 2007) des composants permettant de référencer les données utiles aux preuves. Nous avons nommé ces composants MDA de validation, *unités de preuve*, et *unités d'adaptation*. Nous pensons que la mise en œuvre de ces composants peut participer à un meilleur encadrement méthodologique, une meilleure capitalisation de l'activité de validation des modèles logiciels et une meilleure intégration de la technique de validation dans les processus de développement des modèles. En effet, les unités sont elles-mêmes manipulées en tant que modèles, et sont gérées comme un produit, à part entière, résultant d'un processus d'extraction des données issues de la phase d'analyse. Elles permettent, en tant que cadre conceptuel fédérateur, de capitaliser l'activité et le savoir faire de vérification en regroupant les données nécessaires aux preuves. En conséquence, le processus de développement doit inclure une étape de spécification de l'environnement permettant de générer ensuite des ensembles de comportements finis et ceci de manière complète. Cette hypothèse n'est pas justifiée formellement dans cet article mais repose sur l'idée essentielle que le concepteur ne peut correctement développer un système logiciel que s'il en connaît le périmètre (contraintes, conditions) de son utilisation. Et celui-ci doit lui être fourni formellement comme un résultat du processus d'analyse de l'architecture logicielle conçue, et ceci dans un processus de développement encadré et outillé.

6 Références

Alur R, Dill D (2004). A Theory of Timed Automata. *In Theoretical computer Science*, 126(2) : 183-235.

Bozga M., Graf S., and Mounier L. (2002). IF2.0: A validation environment for component-based real-time systems. *In Proceedings of Conference on Computer Aided Verification, CAV'02, Copenhagen, LNCS. Springer Verlag*.

Blanc X., Sriplakich P., Sadovikh A. (2006). Environnement de développement support à l'ingénierie logicielle guidée par les modèles. *Magazine Génie Logiciel*, No 77.

Clarke T., Evans A., Sammut P., Willians J. (2004). Applied Meamodeling : A foundation for Language Driven Development. *Technical report*, version 0.1, Xactium.

De Belloy S., Dhaussy Ph (2007). CDL : Un langage de description de contextes, syntaxe et sémantique. *Rapport technique, Ensieta (diffusable sur demande)*.

Dhaussy Ph, Boniol F. (2007). Mise en œuvre de composant MDA pour la validation formelle de modèles de système d'information embarqués. *Revue RSTI, numéro spécial Ingénierie des Systèmes d'Information, N°5*.

Dwyers M.B., Avrunin G.S., Corbett J.C. (1999). Patrons in property specifications for finite-state verification. *In Proc. of the 21st Int. Conf. on Software Engineering*, pages 411-420. IEEE Computer Society Press.

Halbwachs N., Lagnier F. and Raymond P. (1993). Synchronous observers and the verification of reactive systems. *In 3rd int. Conf. on Algebraic Methodology and Software Technology (AMAST'93)*.

Hauguen O., Husa K. E., Runde R. K., Stolen K. (2005). Stairs: Towards formal design with sequence diagrams. *In journal of Software and System Modeling*..

Hooman J., Kugler H., Ober I., Votintseva A., Yushtein Y. (2007). Supporting UML-based development of embedded systems by formal techniques. *In journal of Software and System Modeling*.

Janssen W., Mateescu R., Mauw S., Fennema P., Stappen P. (1999). Model Checking for Managers. *Conference Spin'99*, pages 92-107.

Konrad S., Cheng B. (2005). Real Times Specification Patrons. *In Proc. Of the 27th Int. Conf. on Software Engineering (ICSE05)*, St Louis, MO, USA.

Leblanc H., Millan T., Ober I. (2005). Démarche de développement orienté modèles : de la vérification de modèles à l'outillage de la démarche », *in 1ères journées IDM*, page 125-139, Paris.

Muller P. A., Fleurey F., Jézéquel J. M. (2005). Weaving Executability into Object-Oriented Meta-languages. *In S. K. L. Briand (ed), MoDELS 2005, LNCS*, Montego Bay, Jamaica.

Ober Iu., Graf S. and Ober I. (2003). Validating timed UML models by simulation and verification. *In SVERTS'03*, San Francisco, USA.

Object Management Group OMG (2005). Omg unified modeling language specification version 2.0. Technical report.

Roger J. C. (2006). *Exploitation de contextes et d'observateurs pour la vérification formelle de modèles*, thèse de doctorat, Université de Rennes I.

The_Standish_Group, *Chaos Standish Group Internal report*, (1995). The_Standish_Group.

Smith R., Avrunin G.S., Clarke L. and Osterweil L. (2002). Propel : An Approach Supporting Porperty Elucidation. *In Proc. of the 24st Int. Conf. on Software Engineering*, ACM Press, pages 11-21,

Whittle J. (2005). Specifying precise use cases with use case charts. *In MoDELS'06, Satellite Events*, pages 290–301.

Summary

To improve the practice of formal techniques, we propose a concept of "proof context" for representing the environment behavior interacting with a software model and the properties to be checked in this context. The article presents a language named CDL (*Context Description Language*) proposed to the user to describe the proof contexts. Those are exploited, currently in our work, by a technique of model checking based on observer automata. In an MDE approach, the proof contexts are translated into timed automata then into codes exploited by the OBP/IFx (*Observer-Based Prover*) tool for formal analysis. We experienced this approach on some industrial case studies as the formal validation of a AIRBUS-A380 communication protocol. In this paper, we describe an application of our approach to the model validation of an embedded component designed by THALES. The article presents an experience feedback.