# Mise en œuvre de composants MDA pour la validation formelle de modèles de systèmes d'information embarqués

**Philippe Dhaussy* — Frédéric Boniol****

** Laboratoire DTN, ENSIETA, F-29806 Brest cedex 9*
*dhaussy@ensieta.fr*

*** IRIT-ENSEEIHT, 2 rue C. Camichel BP 7122, F-31071, Toulouse cedex 7*
*frederic.boniol@enseeiht.fr*

RÉSUMÉ. *L'article présente la définition et l'exploitation d'un concept d'unité de preuve comme élément encapsulant les données nécessaires à la preuve de propriétés sur un modèle de système à vérifier et plongé dans un environnement. Les modèles des unités, manipulées dans le processus de développement, peuvent être construits sous la forme de modèles abstraits à partir des données fournies par les phases d'analyse et de conception du système. Ces modèles intègrent des contextes de preuve abstraits. Ceux-ci modélisent formellement une exigence (ou une composition d'exigences) de sûreté ou vivacité bornée dans un contexte comportemental donné. Dans une approche IDM, les contextes de preuve sont traduits par transformation en modèles concrets puis en codes exploitables par des outils d'analyse formelle. Nous expérimentons cette approche sur une implantation des unités de preuve intégrant une description des contextes de preuve dans un langage nommé CDL et exploités par un outil OBP (Observer-Based Prover) mettant en œuvre le langage IF et une technique de vérification par observateur.*

ABSTRACT. *The article presents a work in progress about the definition and the exploitation of a concept of proof unit taken into account to encapsulate all the data necessary to the proof of properties on a model composed with an environment. The models of the units, handled in the process of development, can be built, in the form of abstracted models, starting from the data provided by the phases of analysis and system design. These models integrate abstract contexts of proof. Those formally model a safety or bounded liveness property (or a composition of properties) in a context given. In an MDE approach, the contexts of proof are translated into concrete models then into exploitable codes by tools for formal analysis. We experienced this approach by implementation of the proof units integrating a description of the contexts of proof with a language named CDL and exploited by a tool OBP (Observer-Based Prover) implementing the language IF and a technique of checking with observers.*

MOTS-CLÉS : *unités de preuve, vérification formelle, observateur, transformation de modèles.*

KEYWORDS: *proof units, formal checking, observer, model transformation.*

DOI:10.3166/ISI.12.5.133-157 

## 1. Introduction

*Contexte : les systèmes d'information embarqués.* Les systèmes d'information commencent à intégrer les systèmes embarqués pour assurer au sein de ceux-ci des fonctions de plus en plus vitales. On peut citer, dans l'avionique civile, l'exemple du système maintenance embarqué (OMS, pour *On board Maintenance System*) de l'Airbus A380 et du futur A350. L'OMS a pour fonction de collecter en vol l'ensemble des informations de dysfonctionnement, puis en déduire un diagnostic de pannes à partir d'une base d'informations détaillant les modèles de défaillance des équipements de l'avion. L'OMS transmet ensuite en temps réel des rapports de maintenance afin de préparer les opérations de réparation lors de la prochaine escale. Ce système n'est pas critique. Il est en revanche économiquement important pour la compagnie et donc stratégique pour l'avionneur. Son bon fonctionnement permet à la compagnie de réduire les retards dus à des pannes et par suite d'en limiter les coûts. Dans le domaine de l'avionique civile encore, on peut citer également l'exemple du système d'alarmes (FWS, pour *Flight Warning System*) qui analyse en temps réel l'état de fonctionnement de l'avion et prévient l'équipage en cas de panne grave. Afin de ne pas surcharger l'équipage, seules les pannes pertinentes et corrélées à la situation courante de l'avion doivent être remontées. A cet effet, le FWS embarque un modèle de pertinence croisant les pannes et les situations de vol. A l'inverse de l'OMS, le FWS est critique et temps réel. Le non-affichage dans les délais de pannes importantes peut mettre en péril la survie de l'avion.

Les systèmes d'information embarqués sont constitués essentiellement de deux types de composants. D'une part, des composants algorithmiques, associés à une ou des bases d'informations, élaborent de nouvelles données ou corrèlent des données reçues par rapport à des informations précédentes. D'autre part, des composants réactifs, de nature évènementielle, organisent l'enchaînement des actions en fonction des évènements reçus. A titre d'exemple, dans le cas du système OMS, la base de données contenant le descriptif des modèles de défaillances et des diagnostics doit être partagée par plusieurs sous-systèmes : le sous-système de liaisons de données recevant des requêtes du sol et transmettant en retour les informations construites localement par l'avion, l'ensemble des composants de surveillance répartis dans l'avion et renseignant cette base avec les informations observées localement, le système FWS construisant les états de pannes synthétiques à présenter au pilote… Ces sous-systèmes étant parallèles et asynchrones, il est important d'ordonnancer leurs actions et leurs requêtes vis-à-vis de la base d'informations. Il est notamment primordial de démontrer l'absence de blocage et la possibilité pour chaque système d'accéder à cette base dans des délais compatibles avec leurs propres exigences temporelles. Il est également important de démontrer que la levée d'alarmes génère le scénario d'actions et d'interactions prévues entre les différents systèmes de l'avion.

En termes quantitatif, la partie réactive est souvent la plus petite, mais est concernée par la majorité des demandes de corrections issues des essais

(respectivement 40 % pour la taille, et 70 % pour le nombre de corrections dans le cas de systèmes d'information embarqués à bord d'avions d'armes (Ledinot, 2000)). Ces chiffres montrent que, dans le cas des systèmes d'information critiques embarqués, la maîtrise du sous-système réactif reste une difficulté majeure.

***Les techniques formelles.*** Une solution pour répondre à ce défi, explorée depuis plusieurs années par les équipes de recherche, mais aussi par des systémiers comme Airbus ou Dassault Aviation, réside dans l'application des méthodes formelles et en particulier les méthodes de vérification de comportements. De nombreux travaux dans le domaine de la validation formelle ont exploré différents formalismes ou langages (Hooman *et al.*, 2007) et ont contribué à la conception de techniques d'analyse et de preuve exploitables en contexte industriel, et ont ainsi montré leur intérêt et leur puissance. Néanmoins, nous constatons aujourd'hui encore une 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). Il nous apparaît donc indispensable de poursuivre les efforts pour contribuer à une meilleure intégration des techniques formelles dans les processus d'ingénierie.

*Apport de l'IDM : opportunité et composants MDA.* L'ingénierie dirigée par les modèles (IDM) apparaît par ailleurs au niveau industriel comme une réponse potentielle pour la prise en compte de la réalisation des systèmes à logiciels prépondérants qui englobent les systèmes embarqués et les systèmes d'information. Il est donc, à notre avis, pertinent de profiter des réflexions de la communauté IDM pour étudier les conditions d'une meilleure intégration des méthodes formelles. L'IDM n'apporte rien de nouveau, sur un plan théorique, aux concepts d'analyse formelle. Mais elle peut nous permettre, par le biais de la notion de métamodèles et de leur exploitation dans la manipulation et la transformation de modèles, de poser les bases des principes mis en œuvre dans les futurs outils à concevoir. Ceux-ci doivent permettre à l'ingénieur une utilisation plus importante des techniques de vérification et une meilleure capitalisation de son savoir-faire.

En particulier, le concepteur d'un système d'information embarqué doit pouvoir manipuler les exigences et les résultats d'analyse de celles-ci dans des vues spécifiques, compréhensibles par lui, indépendamment des techniques et des langages formels mis en œuvre. Pour conduire son activité de preuve d'exigences ou de propriétés sur un modèle, il doit pouvoir définir l'ensemble des exigences, le comportement de l'environnement dans lequel le système d'information est susceptible d'être plongé durant son exécution et le type de vérification souhaité. Pour mener cette activité, il est nécessaire d'identifier les concepts à introduire dans des métamodèles d'exigences selon certains points de vue d'analyse (performance, sûreté de fonctionnement, dimensionnement...). Il faut également étudier les mécanismes de traduction des modèles d'exigences en exigences de preuve

assimilables par les outils d'analyse formelle. Puis, pour que le concepteur puisse interpréter les diagnostics retournés par ces outils, il faut concevoir des vues qui lui soient compréhensibles et qui seront générées automatiquement.

Au travers des concepts du standard EIA-632 qui définit un ensemble de processus fondamentaux pour la réalisation de systèmes, l'OMG a proposé le métamodèle SPEM (SPEM, 2005) qui permet de spécifier les procédés de développement en UML (*Unified Modeling Language*). Beaucoup d'autres approches de modélisation de procédés (OOSPICE, SMSDM, RUP, MACAO, NEPTUNE, BPMN, BPEL, etc.) ont été proposées. Mais ces méthodes n'ont pas encore complètement intégré, dans un continuum au sens MDE, les composants de validation des modèles produits lors des phases de conception. Il semble donc nécessaire (Leblanc *et al*., 2005) d'étudier cette intégration en étendant les métamodèles en lien avec les méthodes qui ont été jusque-là proposées.

Pour répondre à ce constat, nous étudions dans cet article la formalisation de la notion *d'unité de preuve* comme un élément encapsulant toutes les données nécessaires à la preuve de propriétés sur un modèle de système d'information réactif plongé dans un environnement. L'ensemble des unités de preuve, manipulé par l'utilisateur durant l'activité de validation de son modèle, constitue l'ensemble des obligations de preuve devant être démontrées. Il est également indispensable que ces unités de preuve puissent être manipulées comme des composants à part entière d'un processus de développement, ce qui pose le problème de l'intégration des unités de preuve dans ces processus.

La notion d'unité de preuve, en tant que notion méthodologique pour l'aide au développement et à la validation de la partie réactive d'un système d'information embarqué, constitue le cœur de cet article. L'approche de vérification intégrant cette notion est présentée globalement en section 2. La section 3 formalise ensuite deux notions centrales des unités de preuve, à savoir les notions de contexte et d'observateur. Ces notions sont illustrées et expérimentées sur un cas d'étude décrit en section 4. La notion générale d'unité de preuve est enfin présentée en section 5, en tant que contribution aux approches MDA. Nous concluons sur un premier bilan et des perspectives en section 6.

## 2. Notre approche pour la vérification formelle de modèle

### 2.1. *Vers la notion de contexte d'environnement*

La vérification de propriétés de logiciels temps réels par des techniques de *model-checking* a été largement étudiée ces dernières années. Ceci a conduit à obtenir de nombreux résultats théoriques et permis le développement d'outils qui ont été appliqués à la vérification de systèmes logiciels de taille industrielle. L'approche usuelle repose sur la manipulation d'un modèle, de spécifications (ensemble de propriétés) et de techniques de vérification de propriétés sur ce 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 sont composés avec le modèle à valider ce qui permet de restreindre l'ensemble des exécutions du modèle. 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. C'est l'approche que nous avons choisie d'explorer.

D'un point de vue notation ainsi que méthodologique, la plupart des travaux sur la vérification formelle n'offrent 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). L'idée centrale de l'approche proposée dans cet article est que la modélisation de l'environnement doit au contraire être explicite. En conséquence, le processus de développement doit inclure une étape de construction de spécifications d'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, et ceci dans un processus de développement encadré et outillé.

C'est l'approche explorée dans cet article. Celle-ci repose sur la distinction, dans l'étape de modélisation, entre (a) le modèle du système, (b) le modèle des propriétés à vérifier ainsi que le modèle des comportements non intéressants (mais qui ne sont pas interdits) encodé par des automates observateurs et des restrictions, et (c) le modèle de l'environnement encodé par des automates de contexte. 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 cherchant 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.

Cette approche repose, pour les aspects notations et sémantiques, sur le langage CDL (De Belloy *et al.*, 2007) de description de contextes (*Context Description Language*) décrit section 3.

### 2.2. *Vers la notion de patron de définition de propriétés*

Une autre 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 impossibles à utiliser tel quel. En effet, une exigence peut référencer de nombreux évènements, 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.

### 2.3. *La vérification par observateurs*

L'approche de vérification formelle de propriétés temporisées mise en œuvre repose sur la notion d'automates observateurs (Halbwachs *et al.*, 1993 ; Aceto *et al.*, 1998). Ces automates sont générés automatiquement à partir des propriétés exprimées dans le cadre des patrons de définition et contiennent des états spéciaux appelés états de rejet (*reject*) ou de succès (*success*). L'observateur est construit de manière à encoder une propriété logique et a pour rôle d'observer partiellement les évènements significatifs survenant dans le modèle du système. Lors de la simulation, cet automate est composé, de manière synchrone, au modèle à observer. S'il existe une exécution faisant basculer l'observateur dans l'état de rejet, la propriété encodée par l'observateur sera considérée fausse. Le diagnostic consiste alors en une analyse d'accessibilité des états de rejet sur le graphe d'exécution qui résulte de la composition du modèle à valider et de l'observateur. Dans ce cas, si aucun état de rejet n'est atteint, la propriété est vérifiée. Les observateurs permettent d'exprimer des propriétés de sûreté, d'accessibilité et de vivacité bornée. La technique de

preuve, qui se ramène à une composition[1] et une analyse d'accessibilité[2] de l'état de rejet, est moins coûteuse en calcul que les techniques de *model checking*.

Nous décrivons, dans la suite de l'article, la formalisation des observateurs à l'aide de patrons et le format d'entrée des automates reconnu par notre outil prototype OBP[3].

## 3. La définition des contextes

### 3.1. *CDL : le langage de description des contextes*

Les contextes décrivent le comportement de l'environnement dans lequel le modèle à valider est plongé. Les exigences ou propriétés devant être vérifiées sur le modèle sont associées à la description du contexte. Chaque contexte modélise formellement des exigences à vérifier pour un modèle de système donné dans un contexte comportemental donné. Un contexte caractérise un ensemble d'interactions du modèle avec son environnement, sous la forme de scénarios (ou de comportements) sur lesquels on souhaite restreindre l'activité de vérification. Dans le cas d'un système d'information critique, on peut par exemple s'intéresser à la vérification de propriétés de sûreté (non occurrence d'une situation catastrophique) dans plusieurs classes de scénarios successives : la phase d'initialisation, le mode nominal, un mode dégradé consécutif à une panne… Le comportement de l'environnement avec lequel interagit le modèle, et en particulier l'ensemble des scénarios de cet environnement auxquels on souhaite se restreindre pour mener la preuve, pourrait lui aussi être formalisé sous une forme déclarative (par exemple sous la forme d'un invariant fonction des variables d'états du système à valider et de ses interactions avec l'environnement, interdisant ainsi les configurations entrées/variables d'états falsifiant cet invariant). Nous choisissons une forme impérative, sous la forme de diagrammes d'activités et de séquences. 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 réalisées pour des contextes restreints. Comme nous l'avons précisé précédemment, la description complète du contexte doit être un des résultats du processus de modélisation du système modélisé.

Un langage, nommé CDL (*Context Description Language*), est défini pour décrire le comportement de l'environnement et intégrer les propriétés associées. Précédemment, dans (Roger, 2006 ; Dhaussy *et al.*, 2007), nous avions proposé un

1. La composition est une technique d'assemblage de programmes.
2. L'analyse d'accessibilité correspond à rechercher s'il existe des comportements permettant d'atteindre un état donné.
3. OBP, développé au laboratoire DTN (ENSIETA), est disponible sous licence EPL à l'adresse : http://gforge.enseeiht.fr/projects/obp

langage de contexte exploitant des diagrammes de type UML2, inspirés des *UCC* de (Whittle, 2005). Dans cette première version, ce langage permet 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.

Un modèle *CDL* est structuré en 3 niveaux (figure 1[4]). Au premier niveau, des diagrammes de cas d'utilisation décrivent, par des diagrammes d'activité et de manière hiérarchique, des enchaînements d'activités des entités constituant l'environnement. Les diagrammes de niveau 1 s'interprètent comme un arbre d'exécutions séquentielles ou parallèles. Chaque nœud de cet arbre représente l'un des diagrammes du niveau 2, qui sont des diagrammes de scénarios. Au second niveau, les diagrammes de scénarios acceptent 3 types de nœuds finaux : *ok* (normal), *cancelled* et *stopped*. Chaque nœud du niveau 2 représente un scénario décrit au niveau 3. Les scénarios de troisième niveau sont des diagrammes de séquences UML2.0. 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. Nous précisons cet aspect ainsi que le format de description des propriétés en section 3.2. 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

4. La figure 1 est donnée ici en guise d'illustration des concepts intégrant le métamodèle CDL. La syntaxe concrète du langage n'est pas donnée dans cet article.

restrictions est de préciser les exécutions qui n'intéressent pas l'utilisateur et de permettre de réduire le graphe des exécutions observées en simulation. Les restrictions sont 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 résultat de ce processus. Un métamodèle pour *CDL* a été développé et inclut des éléments de métamodèle correspondant à des patrons de propriétés. 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. Une sémantique des *CDL* est décrite (De Belloy *et al.*, 2007) en termes de traces et s'inspire des travaux de (Hauguen, 2005 ; Roger, 2006).

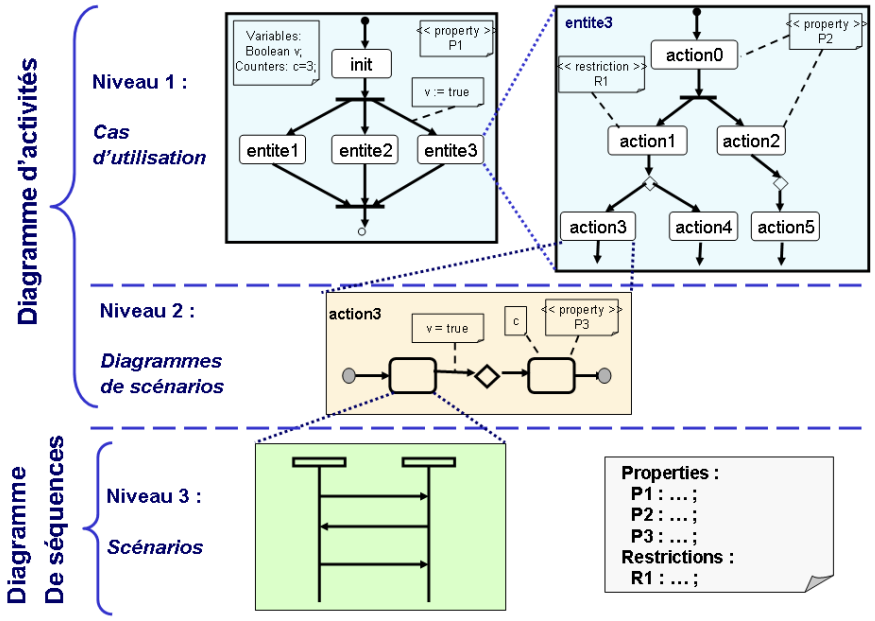


**Figure 1.** *Un modèle CDL (Context Description Language)*

### 3.2. *Spécification des propriétés embarquées dans les contextes*

Pour faciliter la tâche de description des propriétés à vérifier sur les modèles, nous avons repris le principe de leur formulation en s'appuyant sur la notion de patron qui a été proposée dans **(**Dwyers *et al.*, 1999). Ceux-ci capturent, sous forme

textuelle, des types de propriétés usuellement rencontrées dans les documents d'exigences. Chez ces auteurs, les patrons sont classés en familles de base et prennent en compte les aspects temporisés **(**Konrad *et al.*, 2005) des propriétés à spécifier. Les patrons identifiés dans une première approche permettent d'exprimer des propriétés de réponse (*Response*), de prérequis (*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. Les auteurs ont proposé d'identifier la portée 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 évènements d'un ensemble sont concernés par la propriété. Nous donnons, en figure 2, un exemple de propriété de vivacité bornée exprimée avec le patron de type réponse, inspiré de (Smith, 2002). Elle exprime qu'après l'évènement *z* détecté, une occurrence unique de *x* conduit à une occurrence de *y* dans un délai *d*. Nous avons précisé les conditions de prise en compte des occurrences de x et y avec les gardes (*v1 = n*) et (*v2 = m*).

```
Property Pi:
After z
       [
         If (v1 = n) [Exactly one] occurrences of x endif
              [Eventually] leads_to [ <= d ]
              If (v2 = m) [Exactly one] occurrences of y Endif
          y [may occurs before first] x
          [Repeatable]
       ]
```

**Figure 2.** *Un exemple de propriété Pi définie avec le patron de type Response*

Dans le cas des observateurs, l'automate équivalent est montré figure 3. Il contient un état d'erreur ou de rejet (le nœud ☹) en cas de dépassement du délai spécifié ou d'une occurrence de *x* avant celle de *y*.

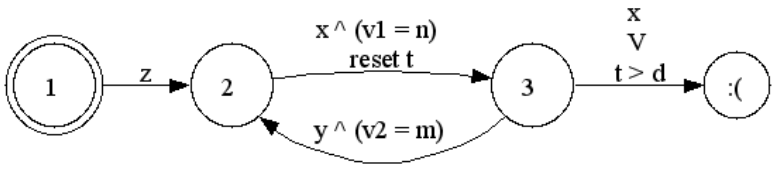


**Figure 3.** *L'automate observateur de la propriété Pi*

En pratique, ces patrons facilitent la définition des observateurs à condition de disposer des algorithmes de génération des formules logiques ou d'automates observateurs équivalents de manière à les exploiter dans les outils de *model-checking*. Dans nos développements, nous avons commencé à développer des programmes de transformations pour les patrons associés à des propriétés de sûreté, d'accessibilité et de vivacité bornée en automates observateurs. Ces programmes de transformation de modèles sont intégrés dans la version de notre outil OBP en cours de développement.

Comme décrit en 3.1, nous lions les propriétés exprimées au contexte décrit dans un modèle CDL et nous en expliquons ici les raisons. 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 dévalidé (*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.

### 3.3. *Vers une méthodologie de définition des propriétés*

Il est nécessaire de proposer aux rédacteurs de documents d'exigences un cadre de spécification des propriétés attendues du système à concevoir, en lien avec la description du contexte d'utilisation de ce système. En ce sens, le couplage entre un modèle de type CDL et des ensembles de propriétés est une voie d'étude à poursuivre. Dans notre approche, la méthodologie suivie pour la description de chaque propriété est : (1) choix d'une classe de patron exprimant l'intention de l'exigence, en prenant en compte les évènements principaux devant être détectés ; (2) ajout des options et des gardes permettant de prendre en compte tous les évènements 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.

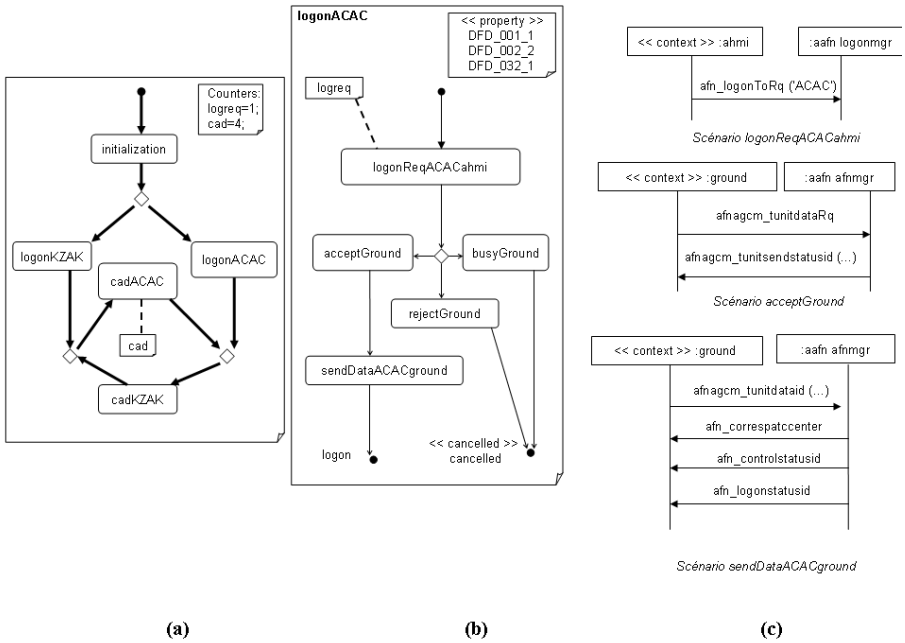


**Figure 4.** *Une vue partielle du modèle CDL pour le cas d'étude AFN*

## 4. Expérimentation outillée sur un cas d'étude

### 4.1. *Un protocole de notification embarqué*

Considérons à nouveau l'exemple du système de maintenance OMS brièvement présenté en introduction. La fonction de ce système étant de transmettre les rapports de pannes collectés à bord, il contient un protocole de communication bord sol. Ce protocole permet l'échange d'informations entre l'avion et un centre de maintenance. Les données transmises sont de type alertes, position, vitesse, etc., et plus généralement des informations sur le contexte applicatif de l'avion (nature et version des équipements embarqués…). Cette communication étant confidentielle compagnie, elle débute par un protocole d'authentification et de notification (AFN pour *Aircraft Facility Notification*) initié par le logiciel de bord à destination de la station sol. Ce protocole a été conçu en langage SDL (ITU-T, 1996) et comporte trois processus : le processus *AfnMgr* gère l'initialisation de l'application et les contacts air/sol. Le processus *LogonMgr* s'occupe de l'identification auprès d'un centre de maintenance. Le processus *CadMgr* prend en charge la gestion des messages échangés.

Dans cette expérimentation, le modèle SDL a été traduit en programme IF 2.0 (*Interchange Format*) (Bozga *et al*., 2002 ; Ober *et al*., 2003), langage conçu par le laboratoire VERIMAG. Cette traduction s'effectue en partie à l'aide du traducteur *sdl2if*. Le modèle peut ainsi être exploité dans l'outil OBP/IFx.

```
Property DFD_002_2 :
Global [
       [Exactly one] occurrences of afn_logonToRq ('ACAC')
       [Eventually] leads_to [ <= 10 ]
       An from
         [Exactly one] occurrences of afnagcm_sendstatus_reject
         [Exactly one] occurrences of afnagcm_sendstatus_busy
         [Exactly one] occurrences of afn_logonstatusid
       End_from
       afn_logonstatusid [cannot occurs before first]
                         afn_logonToRq ('ACAC')
       [Repeatable]
       ]
```

**Figure 5.** *L'exemple de propriété (DFD_002_2) de type Response*

### 4.2. *Le contexte CDL pour ce cas d'étude*

Le modèle CDL, partiellement montré figure 4, a été élaboré à partir, d'une part, des scénarios décrits dans le dossier de conception et formalisé par des diagrammes de séquence et, d'autre part, à partir des exigences dérivées du protocole AFN (CS-SI, 2005). La figure 4 illustre partiellement les 3 niveaux du modèle CDL. Le diagramme de niveau 1 est donné figure 4a. Cette figure décrit le comportement du

système dans son environnement par un enchaînement de modes. Le diagramme de scénarios *logonACAC* donné figure 4b détaille le mode de même nom et décrit ce dernier comme un enchaînement d'activités débutant par un scénario de requête d'authentification (*logonReq*), et se terminant soit par un *logon*, soit par un *cancelled*. Enfin, les scénarios liés au mode *logonACAC* sont donnés figure 4c. Ce mode est soumis par ailleurs à plusieurs exigences dont, par exemple, la propriété *DFD_002_2*. L'expression de cette exigence est donnée figure 5. Cette exigence est une propriété de réponse en temps bornée imposant :

– d'une part, qu'après toute occurrence du message, *afn_logonToRq('ACAC')* sera reçue une occurrence dans un délai maximum de 10 secondes d'un des trois messages *afnagcm_sendstatus_reject*, afnagcm_sendstatus_busy, ou afn_logonstatusid ;

– et, d'autre part, qu'il est impossible de recevoir afn_logonstatusid avant tout requête *afn_logonToRq('ACAC')*.

Cette exigence est traduite selon un observateur contenant un état de rejet dont l'accessibilité exprime la falsification de la propriété. Cet observateur est donné figure 6. Plusieurs propriétés significatives ont été vérifiées par cette technique (voir (Roger, 2006) pour la description détaillée de ce cas d'étude).

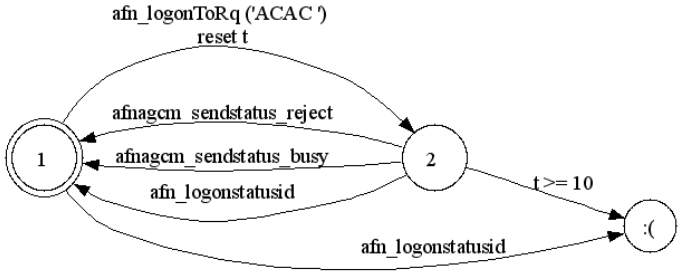


**Figure 6.** *L'observateur de la propriété DFD_002_2*

### 4.3. *Outil prototype OBP/IFx*

Pour mener nos expérimentations, nous nous sommes basés sur le langage d'implantation formel IF 2.0. Celui-ci repose sur le formalisme étendu des automates temporisés (Alur *et al*., 1994) adapté au contexte des systèmes communicants de manière asynchrone. Le laboratoire VERIMAG a développé un outillage (*IFx*) permettant la simulation de programmes IF et la génération des graphes d'accessibilité. Autour de cet outil, nous avons développé un outil nommé *OBP* (figure 7) qui intègre IFx. L'objectif de ce développement est de disposer d'une plateforme d'expérimentation pour mieux appréhender les possibilités et les points durs dans l'exploitation de techniques de *model-cheking*.

Dans la configuration actuelle, les modèles CDL sont édités dans un environnement Eclipse et importés dans OBP au format XMI[5]. L'interprétation d'un modèle CDL permet de générer une description du comportement de l'environnement dans un format interne basé sur le langage 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. L'outil génère également les automates observateurs et de restriction à partir des propriétés et des restrictions qui sont décrites dans le modèle CDL.

Les modèles à valider dans OBP sont importés, quant à eux, actuellement au format IF 2.0. Pour prendre en compte les imports de modèles au format type UML2, AADL (SAE, 2004), SDL, il est encore nécessaire de mettre en œuvre des traducteurs adéquats. Dans ce but, des développements sont actuellement en cours en intégrant OBP dans une plateforme de développement de modèle conçue dans l'univers Eclipse et intégrant des programmes de transformation de modèles au format KerMeta.

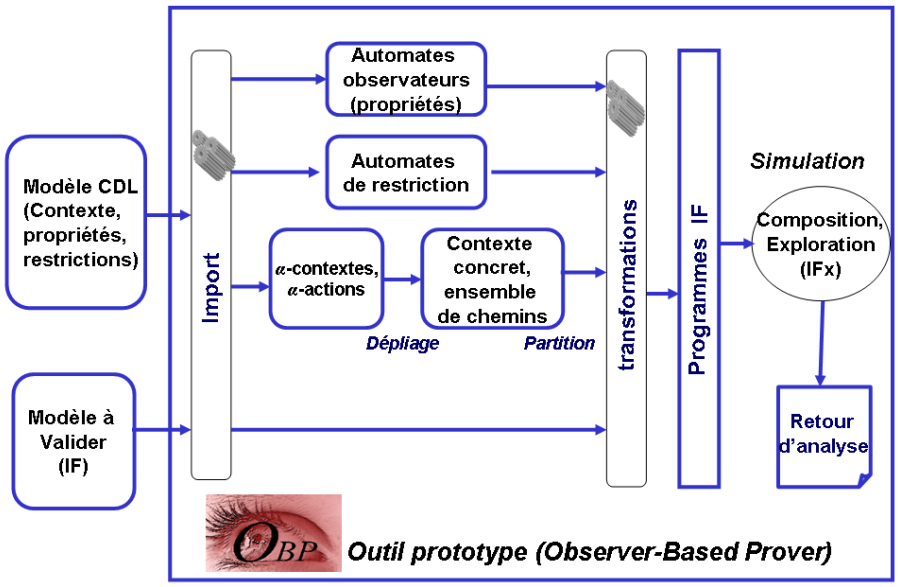


**Figure 7.** *L'outil prototype OBP/IFx*

Dans notre prototype, les modèles CDL sont traduits en *α-contextes.* Ceux-ci sont destinés à être raffinés ou *dépliés* en des automates temporisés.

5. XML (eXtensible Markup Language) *Metadata Interchange*.

La traduction des *α-contexte*s implique plusieurs phases automatisées :

– le dépliage d'un *α-contexte* en un automate nommé *contexte concret*. 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,

– le partitionnement de l'automate déplié 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 traduisibles en langage IF,

– enfin, chaque chemin généré est transformé en un automate IF et composé avec le système et les automates observateurs.

L'analyse d'accessibilité est réalisée sur le résultat de cette composition. S'il existe un état atteint *reject* d'un observateur de propriété pour l'un des chemins, alors la propriété n'est pas vérifiée. L'intérêt de la composition du modèle et de ses observateurs avec chaque chemin est de restreindre fortement le graphe d'exécution généré. Une limite pour cette approche peut être liée à une explosion du nombre de chemins générés dans les cas où l'exécution de l'environnement est complexe. Là aussi, le processus, encadré et outillé, de modélisation du système doit permettre de décrire et de partitionner l'ensemble des scénarios du contexte. Cette problématique n'est pas développée dans le cadre de cet article.

### 4.4. *Discussion*

Nous avons décrit ici une technique de preuve reposant sur la description d'observateurs et de contextes. Pour la validation d'un système réel, l'ensemble des exigences à vérifier est en général important. L'outil prototype OBP nous permet aujourd'hui de confronter l'approche sur deux cas industriels significatifs de systèmes d'information embarqués : le protocole AFN, sous-système d'un système avionique, et un système de contrôle et de commande. L'activité de validation des modèles implique, dans les cas traités la manipulation de nombreux contextes et exigences et un enchaînement des preuves.

Ce premier retour d'expérience nous incite à aller plus loin dans l'établissement de la méthodologie. 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 un composant permettant de référencer les données utiles aux preuves. Nous nommons ce composant MDA de validation, *unités de preuve*, que nous décrivons dans la section suivante.

## 5. Un composant MDA : l'unité de preuve

La communauté IDM entreprend aujourd'hui des travaux concernant la modélisation des processus de développement, prenant en compte toutes les phases et activités liées au développement d'un système logiciel. Dans le cadre de la validation formelle de modèles, l'objectif est d'intégrer au mieux ces activités dans le processus de développement. Le projet OMEGA[6] en est un bon exemple où les motivations visent une forte intégration des techniques de méthodes formelles dans les processus. Les moyens mis en œuvre dans ce projet reposent sur le couplage d'outils commerciaux UML (*via* le format XMI) et des outils de preuve de type *model-checker* et *theorem prover*. Un des résultats importants de ce projet est la traduction d'un noyau UML en langages formels adaptables aux outils de vérification. Ce point est essentiel pour que l'utilisateur n'ait pas à manipuler les programmes formels de bas niveau. Un autre résultat est la proposition de capture des exigences, par exemple par des LSC (Damm, 2001), permettant à l'utilisateur d'exprimer facilement les propriétés devant être vérifiées sur son modèle. Nous pourrions citer d'autres propositions qui ont toutes pour objectif l'intégration de méthodes et de techniques qui, jusque-là, étaient considérées séparément et donc difficilement intégrables ensemble.

Pour les techniques de preuve formelles, l'approche IDM est une opportunité permettant d'embarquer dans des modèles productifs, c'est-à-dire interprétables par des programmes (ou d'autres modèles), les notions ou méta-entités définissant les objets manipulés durant les preuves. Au regard des propositions autour du standard EIA-632 et sa déclinaison sous forme de métamodèle SPEM de l'OMG, il est pertinent de formaliser les processus de validation formelle par la construction de modèles de processus spécifiques aux activités de vérification. Ceux-ci doivent incorporer les données nécessaires aux preuves. Dans cet objectif, nous avons proposé (Dhaussy *et al.*, 2007) la notion *d'unité de preuve* comme un composant structurant et référençant les données nécessaires aux preuves. Un composant de ce type doit être considéré comme un objet élémentaire, exploité par le modèle de processus de validation. Celui-ci manipule, pour un modèle donné soumis à validation et un contexte donné, un ensemble d'unités de preuve référençant l'ensemble des exigences et des scénarios à considérer pour ce modèle. En tant que composant MDA, il embarque non seulement les références aux modèles (contexte, propriétés, modèles à valider), et les métamodèles associés, mais aussi l'ensemble des règles de transformation permettant d'interpréter ce composant pour la génération des programmes d'analyse formelle, le déclenchement de ces analyses et la construction des données de retour d'analyse. Nous détaillons ici le rôle et la structure des unités de preuve.

6. http://www-omega.imag.fr

### 5.1. *Rôle et objectifs des unités de preuve*

L'objectif recherché est de formaliser des composants de validation manipulés par l'utilisateur dans un processus de développement. L'idée sous-jacente est de contribuer à outiller l'activité de validation formelle dans le but de mieux l'encadrer. L'approche IDM conduit à élaborer des modèles structurant les données nécessaires aux techniques de vérification de propriétés. Ces modèles capturent la logique métier (Blanc *et al*., 2006) qui supporte les activités de preuve indépendamment des outils de vérification utilisés. Des transformations adéquates devront traduire ces modèles en des modèles dépendants des outils de vérification (ou *model-checkers*), en prenant en compte leurs spécificités et leurs contraintes.

L'avantage de cette approche est un meilleur accès aux technologies de vérification et une meilleure productivité. L'utilisateur manipule des entités du même niveau que les entités de son modèle de conception. Il n'a pas à se préoccuper des détails de mise en œuvre des preuves comme par exemple la manipulation des formules de logiques temporelles, des systèmes de transitions ou des hypothèses manipulées à bas niveau pour restreindre l'explosion combinatoire. Il doit pouvoir manipuler les modèles de vérification et générer des simulateurs et analyseurs spécifiques comme il utilise les compilateurs pour générer du code exécutable pour des plateformes spécifiques. Un autre avantage est de pouvoir capitaliser dans les modèles son activité de vérification et son savoir-faire, comme cela peut être fait de manière informelle dans certains processus industriels sous la forme de plan de test. Les modèles, indépendants des outils de preuve, devront être plus pérennes et être transformés ensuite en des modèles et programmes formels spécifiques aux différents outils de vérification, basés sur des formalismes comme, par exemple, les automates temporisés, ou les outils de type TINA (Berthomieu *et al*., 2006) exploitant les réseaux de Petri, ou ceux exploitant les algèbres de processus. Cette démarche peut s'étendre à d'autres langages, type Altarica (Arnold, 2000), dans le cas, par exemple, d'analyses de sûreté de fonctionnement.

### 5.2. *Structure des unités de preuve*

Nous définissons la notion *d'unité de preuve* comme un concept, de niveau utilisateur, englobant les données nécessaires à l'activité de preuve en vue de la validation d'un modèle. Plus précisément, une unité de preuve fait référence (figure 8) à plusieurs entités que sont les contextes de preuve, les métamodèles et modèles impliqués dans la construction des preuves, les transformations de modèles en vue de générer les codes assimilables par les prouveurs et, enfin, les entités de représentation des résultats des preuves.

En d'autres termes, les modèles d'unités de preuve sont des modèles abstraits, conformes à un métamodèle. Ces modèles abstraits sont ensuite traduits par transformation en modèles concrets pris en charge par des outils d'analyse formelle.

Le métamodèle définissant les unités de preuve est composé des méta-entités qui définissent les éléments les constituant et que nous détaillons ici.

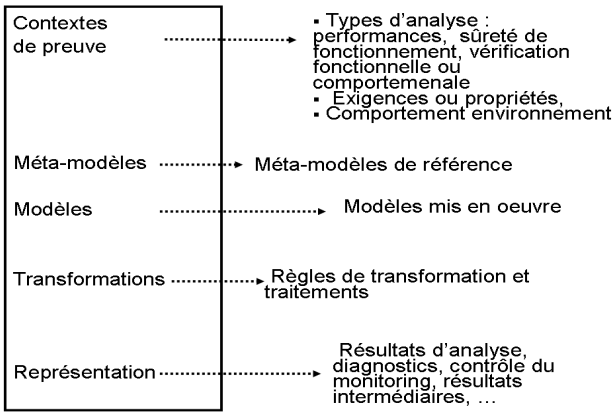


**Figure 8.** *Le concept d'unité de preuve*

Les *contextes de preuve* regroupent les données caractérisant les exigences ou propriétés devant être vérifiées, le comportement de l'environnement dans lequel est plongé le modèle et le type d'analyse souhaitée par l'utilisateur. Nous avons montré, en section 3, une formalisation impérative du comportement de l'environnement sous la forme de modèle CDL pour décrire le comportement de l'environnement et les propriétés qui sont à vérifier.

Dans un contexte d'IDM, une unité de preuve fait également référence aux *métamodèles* et les *modèles* mis en œuvre dans l'activité de validation. Ceux-ci sont impliqués dans le processus de construction des preuves, ou plus précisément dans la construction des prouveurs ou des simulateurs qui permettront les analyses sur le modèle. Les modèles à valider référencés sont manipulés par l'utilisateur dans des formats, par exemple, du type UML, SDL, AADL, etc. Les modèles issus automatiquement des transformations seront produits dans des formats de bas niveau comme ceux reposant, par exemple, sur les automates temporisés, les réseaux de Petri, les algèbres de processus, etc. Les métamodèles de ces langages doivent donc être référencés par l'unité de preuve. Les modèles formels n'ont pas pour objectif d'être manipulés explicitement par l'utilisateur mais doivent rester « cachés »[7].

Pour pouvoir générer les prouveurs (les simulateurs d'analyse), des *transformations* de modèles sont nécessaires pour traduire, d'une part, les modèles des contextes de preuve et, d'autre part, les modèles de conception en des

7. Par analogie aux techniques de compilation de programmes, un code assembleur n'a pas vocation à être explicitement manipulé par l'utilisateur.

programmes formels exploitables par ces outils de preuve ou de simulation. L'ensemble des règles de transformations est référencé dans une unité de preuve.

Enfin, en vue de pouvoir présenter à l'utilisateur les résultats des analyses réalisées, un modèle de *représentation* ou de diagnostic est référencé dans l'unité de preuve. L'interprétation de ce modèle doit permettre à l'utilisateur non seulement de détecter mais aussi de reconnaître et comprendre le type de défaillance d'une propriété et le composant du modèle à incriminer. Ce modèle doit être structuré de manière à être le réceptacle des données générées par les analyseurs lors de l'exécution des preuves. Des transformations doivent permettre de construire des modèles de données interprétables dans le modèle de l'utilisateur contenant les informations nécessaires au diagnostic. Ces données exploitables par l'utilisateur donneront souvent lieu à une visualisation. Dans notre approche, la mise en œuvre des modèles de représentation doit encore faire l'objet de développements spécifiques.

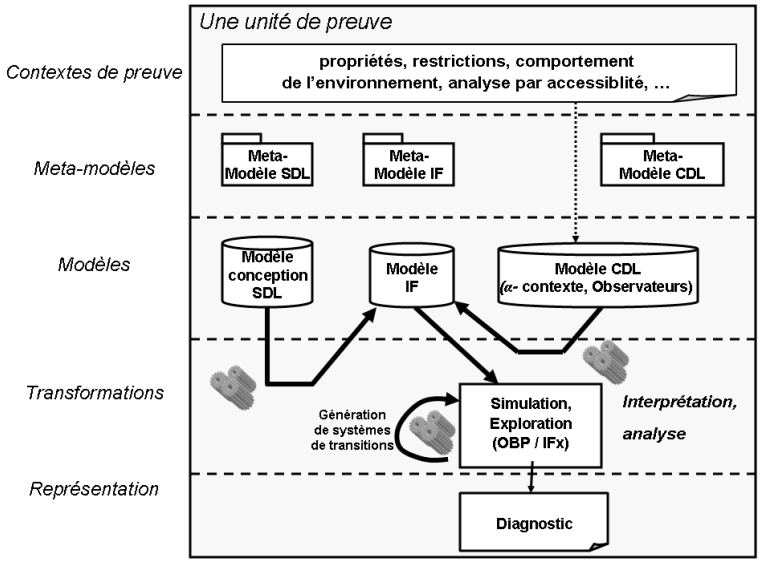


**Figure 9.** *Une unité de preuve pour le cas d'étude*

### 5.3. *Un exemple d'unités de preuve pour le cas d'étude*

La figure 9 schématise d'une manière informelle la structuration d'une unité de preuve dans le cas de la validation du modèle SDL[8] du protocole AFN décrit en 4.1.

8. La schématisation de l'unité de preuve aurait pu être faite sur le même principe dans les cas de modèles de conception UML ou AADL.

Dans cet exemple, les contextes de preuve référencent des modèles CDL. La mise en œuvre des preuves est réalisée *via* la génération de programmes formels IF et repose sur la technique de vérification par observateurs. Les transformations traduisent, d'une part, les modèles de conception SDL et, d'autre part, les modèles CDL en modèles IF. L'outil OBP/IFx transforme par simulation les modèles IF en systèmes de transitions.

### 5.4. *Une perspective pour la génération des unités de preuve*

L'ensemble des unités de preuve, manipulé par l'utilisateur durant l'activité de validation de son modèle, constitue l'ensemble des obligations de preuve devant être démontrées et les conditions dans lesquelles les preuves s'opèrent. Dans notre approche, nous envisageons que de telles unités de preuve peuvent être construites (figure 10) par transformation de modèles, sous la forme de modèles, à partir des données fournies par les activités d'analyse et de conception du système. Les activités amont d'analyse et de conception génèrent des modèles qui sont eux-mêmes conformes à des métamodèles qui capturent l'ensemble des méta-entités manipulées par l'utilisateur lors de ces activités.

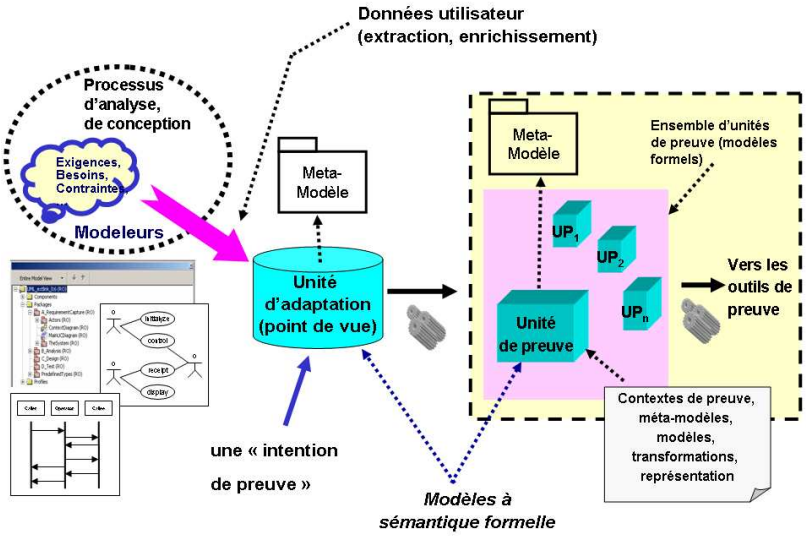


**Figure 10.** *Les unités d'adaptation et de preuve*

Dans un processus de conception d'un système, les activités de définition des exigences techniques et de la solution s'enchaînent de façon récursive, à différents niveaux d'abstraction. Les exigences techniques se déclinent en termes d'entrées/sorties, de descriptions d'états, de configurations, de profils opérationnels

d'utilisation, d'interfaces, de contraintes, etc. Dans le cas général, la définition de la solution correspond à des vues abstraites et enrichit les exigences par l'apport d'exigences dérivées liées à la solution.

Les modèles de cas d'utilisation décrivent les relations entre les entités externes, en particulier, les autres systèmes en interaction avec le système, et les cas d'utilisation du système logiciel. Ils détaillent, pour chaque cas d'utilisation, les interactions entre les entités externes et le système par des scénarios représentatifs des conditions d'utilisation (Booch *et al*., 2003). Ceux-ci sont décrits, par exemple, sous la forme de diagrammes de séquences UML, de scénarios de ceux du type issus d'outils de simulation, par exemple, de modèles SDL, etc. La dynamique du comportement de l'environnement, des collaborations de ses composants et du système est modélisée sous la forme de graphes d'activité.

Dans une démarche outillée dirigée par les modèles, et de notre point de vue, les phases d'analyse et de conception fournissent donc un ensemble de modèles du système et de son environnement qui doivent permettre de construire, avec l'aide de l'utilisateur, ces unités d'adaptation. Celles-ci donneront lieu, par transformation, à la construction des ensembles d'unités de preuve. Nous devons encore poursuivre le travail en cours de définition formelle des unités d'adaptation et des règles de transformation pour générer les ensembles d'unités de preuve.

## 6. Conclusion

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. L'approche choisie repose sur la manipulation d'unités de preuve exploitées dans le processus de développement, offrant un cadre à l'utilisateur pour mener l'activité de validation des modèles qu'il manipule. Chaque unité englobe les données nécessaires à la preuve. Elle incorpore, dans un contexte de preuve, la définition du comportement de l'environnement dans lequel le modèle du système est plongé durant l'analyse, ainsi que les propriétés à vérifier. La notation CDL décrite, intégrant un formalisme de description des contextes ainsi que les patrons de définition de propriétés, est donnée, dans cet article, en tant que langage prototype.

Les expérimentations de vérification de propriétés, menées sur des modèles de systèmes d'information, dans des contextes différents, permettent d'entrevoir l'intérêt d'une telle approche. D'une part, elles contribuent à une meilleure intégration de la technique de validation utilisée dans le 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. D'autre part, le métamodèle et le caractère exécutable des modèles d'unités de preuve, ainsi que les transformations mises en œuvre contribuent à mieux formaliser, à un niveau d'abstraction donné, les stratégies de preuve d'exigences, indépendamment des langages et outils mis en œuvre à des

niveaux plus concrets. Enfin, les unités de preuve, en tant que cadre conceptuel fédérateur, permettent de capitaliser l'activité et le savoir-faire de vérification en regroupant les données nécessaires aux preuves.

Ces travaux seront prolongés dans trois directions. La première concerne le développement d'un prototype de manipulation d'unité de preuve intégrant l'outil OBP mis en œuvre durant ce travail qui doit se poursuivre pour pouvoir valider la démarche sur des applications d'une complexité supérieure. L'approche décrite ici étant indépendante de la technique de vérification, il est nécessaire pour atteindre cet objectif de poursuivre la formalisation du concept d'unité de preuve en l'interfaçant avec d'autres techniques d'analyse formelles. Les techniques envisagées exploitent soit la vérification formelle outillée par des *model-checkers* tels que SPIN reposant sur le langage Promela (Holzmann, 2003), TINA (Berthomieu *et al.*, 2006) exploitant les réseaux de Petri, soit des utilitaires de manipulations de systèmes de transitions. L'interfaçage des unités de preuve avec d'autres techniques d'analyse, adressant d'autres types de comportement, tels la sûreté de fonctionnement ou les aspects temps réel, est également envisagé. Dans le premier cas, on parlera d'unité de preuve pour la sûreté de fonctionnement que l'on interfacera avec un langage formel tel que AltaRica (Arnold, 2000). Dans le second cas, on parlera d'unité de preuve pour le temps réel et l'on s'orientera vers des outils formels d'ordonnançabilité tels que Cheddar (Singhoff, 2004).

La seconde voie de recherche consiste à explorer, en amont du concept d'unité de preuve, un processus d'ingénierie permettant de construire et synthétiser les unités d'adaptation. Si les unités de preuve constituent un cadre précis pour la capitalisation des informations nécessaires à une analyse selon un point de vue donné, la difficulté est désormais repoussée en amont sur la façon de construire ces unités de preuve. A cette fin, la notion d'unité d'adaptation est envisagée pour permettre la génération des unités de preuve. Reste en revanche la construction ou le renseignement de ces unités d'adaptation à partir des informations d'architecture et de conception du système développé (le système d'information OMS par exemple). Il est vraisemblable qu'une telle construction ne puisse pas être complètement automatisée ni formalisée. En revanche, il sera nécessaire d'étudier et de proposer un processus d'ingénierie, outillé, permettant de guider et d'encadrer la construction de ces unités d'adaptation. L'étude de ce processus constitue la seconde piste d'investigation envisagée après ce travail.

Enfin, à l'instar des propriétés, la manipulation des contextes reste une tâche délicate, et ce malgré leur formalisation. Notamment, la représentation d'un environnement complet par un seul contexte peut, dans les cas complexes, conduire ici encore à un phénomène d'explosion des comportements à modéliser. Pour pallier cette difficulté, la troisième piste ouverte par les travaux présentés dans cet article porte sur la notion de composition de contexte et par suite sur la notion de preuve (et de couverture de preuve) faite sur une composition de contexte. Ces deux dernières pistes feront l'objet de travaux de recherche conjoints entre l'ENSIETA et l'IRIT-ENSEEIHT d'une part, et AIRBUS, CS-SI et THALES d'autre part.

## 7. Bibliographie

Aceto L., Bouyer P., Burgueno A., Larsen K.G., "Model checking *via* reachability testing for timed automata", Bernhard Steffen (ed.), *TACAS'98*, vol. 1384 of Lecture Notes *in Computer Science*, Springer-Verlag, 1998, p. 263-280.

Alur R., Dill D., "A Theory of Timed Automata", *Theoretical computer Science*, vol. 126, n° 2, 25 April 1994, p. 183-235.

Arnold A., Griffault A., Point G., Rauzy A., "The AltaRica formalism for describing concurrent systems", *Fundamenta Informaticae*, 40, 2000, p. 109-124.

Blanc X., Sriplakich P., Sadovikh A., « Environnement de développement support à l'ingénierie logicielle guidée par les modèles », *Magazine Génie Logiciel*, n° 77, Juin 2006.

Berthomieu B., Vernadat F., "Time Petri nets analysis with TINA", *3rd Int. Conf. on the Quantitative Evaluation of Systems (QEST'2006)*, Riverside (USA), 11-14 Septembre 2006, p. 123-124.

Booch G., Kroll P., Krutchen P., *The Rational Unified Process made easy: A practitioner's guide to the RUP*, Addison-Wesley Professional, Avril 2003.

Bozga M., Graf S., Mounier L., "IF-2.0: A validation environment for component-based real-time systems", *Proceedings of Conference on Computer Aided Verification, CAV'02, Copenhagen, LNCS*, Springer Verlag, June 2002.

CS-SI, Document de fonctions dérivées du protocole AFN, Rapport technique, CS-SI, Airbus, 2005.

Damm W., Harel D., "LSCs: Breathing life into message sequence charts", *Formal Methods System Des.*, vol. 19, n° 1, 2001, p. 45-80.

De Belloy S., Dhaussy Ph., CDL : Un langage de description de contextes, syntaxe et sémantique, Rapport technique, Ensieta, Juillet 2007.

Dhaussy Ph., Roger J.C., Boniol F., « Mise en œuvre d'unités de preuve pour la vérification formelle de modèles », *Conférence IDM'07*, 29-30 mars 2007, Toulouse, p. 101-116.

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

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

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

Holzmann G.J., *The SPIN MODEL CHECKER, Primer and Reference Manual*, Addisson-Wesley, Septembre 2003.

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

ITU-T, Languages for telecommunications applications – specification and description language (SDL), ITU-T Recommendation Z.100, 1996.

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

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

Leblanc H., Millan T., Ober I., « Démarche de développement orienté modèles : de la vérification de modèles à l'outillage de la démarche », *1re journées IDM*, Paris, Juin 2005, p. 125-139.

Ledinot E., « Regard sur 10 années d'introduction des methods formelles. Approches Formelles dans l'Assistance au Développement de Logiciels », *AFADL'2000*, Grenoble, janvier 2000.

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

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

Obj, *Software Process Engineering Metamodel (SPEM) 1.1 Specification*, janvier 2005, formal/05-01-06.

Roger J. C., Exploitation de contextes et d'observateurs pour la vérification formelle de modèles, Thèse de doctorat, université de Rennes I, décembre 2006.

SAE (Society of Automotive Engineers), "The SAE Architecture Analysis & Design Language Standard Information Website", http://www.aadl.info/, 2004.

Singhoff F., Legrand J., Nana L., Marce L., "Cheddar: a flexible real time scheduling framework", *Ada Letters*, vol. 24 n° 4, 2004.

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

Whittle J., "Specifying precise use cases with use case charts", *MoDELS'06, Satellite Events*, 2005, p. 290-301.