

Contribution à la formalisation de contextes et d'exigences pour la validation formelle de logiciels embarqués

Philippe Dhaussy¹, Pierre-Yves Pillain¹, Stephen Creff¹, Amine Raji¹,
Yves Le Traon² et Benoit Baudry³

¹ UEB, Laboratoire LISyC, ENSIETA, BREST, F-29806 cedex 9
{dhaussy, pillai, creffst, rajiam}@ensieta.fr

² Université du Luxembourg, Campus Kirchberg
Yves.letraon@Uni.lu

³ Equipe Triskell, l'IRISA, RENNES, F-35042
bbaudry@IRISA.fr

29 mars 10

Résumé. Un défi bien connu dans le domaine des méthodes formelles est d'améliorer leur intégration dans les processus de développement industriel. Dans le contexte des systèmes embarqués, l'utilisation des techniques de vérification formelle nécessitent tout d'abord de modéliser le système à valider, puis de formaliser les propriétés devant être satisfaites sur le modèle et enfin de décrire le comportement de l'environnement du modèle. Ce dernier point que nous nommons « contexte de preuve » est souvent négligé. Il peut être, cependant, d'une grande importance afin de réduire la complexité de la preuve. Dans notre contribution, nous cherchons à proposer à l'utilisateur une aide pour la formalisation de ce contexte en lien avec la formalisation des propriétés. Dans ce but, nous proposons et expérimentons un langage (DSL), nommée CDL (*Context Description Language*), pour la description des acteurs de l'environnement, basée sur des diagrammes d'activités et de séquence et des patrons de définition des propriétés à vérifier. Les propriétés sont modélisées et reliées à des régions d'exécution spécifiques du contexte. Nous illustrons notre contribution sur un exemple et décrivons des résultats sur plusieurs applications industrielles embarquées.

Mots-clés : Contextes de preuve, exigences formelles, modèles de contexte, observateurs, automates temporisés, vérification.

1 Introduction

Dans le domaine des systèmes embarqués, les architectures logicielles doivent être conçues pour assurer des fonctions critiques soumises à des contraintes très fortes en termes de fiabilité et de performances temps réel. En raison de ces contraintes, les architectures logicielles embarquées sont soumises à un processus de certification qui nécessite un développement très rigoureux. Toutefois, en raison de la complexité croissante des systèmes, leur conception reste une tâche difficile.

Dans le but d'accroître la fiabilité des logiciels, les méthodes formelles contribuent à l'apport de solutions rigoureuses et puissantes pour aider les concepteurs à produire des systèmes non défaillants. À cet effet, les méthodes de vérification formelle de comportement de modèles ont été explorées depuis plusieurs années par de nombreuses équipes de recherche [2, 8], et expérimentées par des industriels. Mais aujourd'hui, les logiciels embarqués intègrent des fonctionnalités de plus en plus complexes ce qui rend difficile la mise en œuvre de ces méthodes. Malgré la performance croissante des outils de *model-checking*, leur utilisation reste difficile en contexte industriel. Leur intégration dans un processus d'ingénierie industriel est encore trop faible comparativement à l'énorme besoin de fiabilité dans les systèmes critiques. Cette contradiction trouve en partie ses causes dans la difficulté réelle de mettre en œuvre des concepts théoriques dans un contexte industriel.

Les techniques de vérification formelle souffrent aussi du problème bien identifié de l'explosion combinatoire du nombre de comportements des modèles, induite par la complexité interne du logiciel qui doit être vérifiée. Cela est particulièrement vrai dans le cas des systèmes embarqués temps réel, qui interagissent avec des environnements impliquant avec un grand nombre d'acteurs. Un moyen pour le réduire est de spécifier et de restreindre le comportement de l'environnement du système dans lequel il est plongé. Le système est ensuite synchronisé fortement avec son environnement lors du processus de vérification. Ce contexte correspond à des phases opérationnelles bien identifiées, comme, par exemple, l'initialisation, les reconfigurations, les modes dégradés, etc. Aussi, la vérification des propriétés sur les modèles d'un système nécessite leur expression formelle dans des formalismes de type logique temporelle comme LTL [18] ou CTL [16]. Bien que ces langages aient une grande expressivité, ils ne sont pas faciles à utiliser par des ingénieurs lors de leurs projets. Pour surmonter ce problème, certaines approches [5, 12, 10] ont été proposées pour formaliser les propriétés temporelles à l'aide de patrons d'expression plus proche des types de langages qu'ils ont l'habitude de manipuler.

Face à ce constat, nous avons proposé [21, 22] de définir un cadre structurant permettant à l'utilisateur de décrire formellement des *contextes de preuves* au travers de l'utilisation du langage de description de contextes, CDL (*Context Description Language*). Ce DSL¹ permet de spécifier des contextes sous la forme de scénarios et des propriétés temporelles à l'aide de patrons de définition de propriété. De plus, CDL offre la possibilité à l'utilisateur d'associer chaque propriété à une phase du comportement de l'environnement du système, c'est-à-dire un contexte spécifique. Les contextes de preuves sont ensuite exploités pour générer automatiquement des programmes formels assimilables par des outils de vérification.

Dans cet article, nous rendons compte d'un retour d'expérience sur l'application de notre approche et ce langage sur plusieurs cas d'étude du domaine aéronautique. Nous présentons l'approche et décrivons les résultats sur des expérimentations de mise en œuvre de preuves formelles par des ingénieurs. Nous montrons, tout d'abord, l'intérêt de spécifier le contexte dans lequel le système sera utilisé lors de la simulation en vue de réduire le problème de l'explosion combinatoire. Ensuite, nous montrons comment formaliser, avec CDL, les contextes et les propriétés et ensuite comment relier ces propriétés à des régions spécifiques du contexte.

Pour faciliter la compréhension de notre approche, nous l'illustrons par un exemple simple déduit d'un cas industriel réel (le logiciel du système S_CP^2 montré en figure 1). S_CP contrôle les modes internes d'un système relié à des périphériques (radars, capteurs, effecteurs...) et répond à des signaux provenant d'acteurs de l'environnement (HMI , $Device$).

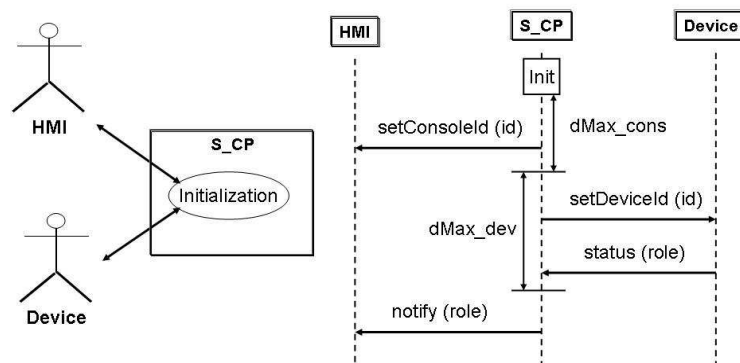


Figure 1 : Le système S_CP : un diagramme de séquences et un cas utilisation (partiel) décrivant le comportement du système et son environnement lors de la phase d'initialisation

¹ Domain Specific Language

² Pour des raisons de confidentialité, les noms de la société partenaire et du système ne sont pas mentionnés dans ce document.

L'article est organisé comme suit : le paragraphe 2 décrit le périmètre de nos travaux dans le domaine de la pratique actuelle des techniques de vérification formelle et présente des travaux connexes. Le paragraphe 3 décrit le langage CDL pour la spécification des contextes et des propriétés. La méthodologie proposée utilisée pour les expérimentations est présentée en partie 4, ainsi que l'outillage mis en œuvre. En partie 5, nous donnons quelques résultats³ sur plusieurs études de cas industriels. Enfin, nous concluons, en partie 6, par une discussion sur notre approche et des travaux futurs.

2 Objectifs et travaux connexes

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 tels quels. 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. Ce constat est illustré dans un exemple d'exigence *R* (listing 1) extrait du cahier de charge du système *S_CP*. On peut constater que la formalisation de *R* évoque beaucoup d'événements et d'actions opérées par le système et l'environnement, un historique d'exécution. Due à sa formalisation ambiguë, son interprétation peut très vite poser problème. Le recours à des formules d'une logique temporelle n'est pas aisé car leurs expressions peuvent être d'une grande complexité. Elles demandent beaucoup d'expertise, ce qui les rend difficilement lisibles ou manipulables par des ingénieurs.

Exigence *R* :

« Au cours de la procédure d'initialisation, *S_CP* doit associer un identificateur à chaque périphérique (*Device*) du système, avant une unité de temps *dMax_dev*. Il doit également associer un identificateur pour chaque console (*IHM*), avant une unité de temps *dMax_cons*. *S_CP* transmet un message *notifyRole* pour chaque périphérique et chaque console connectés. *S_CP* transmet un message *notifyRole* à chaque périphérique connecté et chaque console connectée. L'initialisation s'achève avec succès, lorsque *S_CP* a affecté tous les identificateurs de périphérique et de console et lorsque tous les messages *notifyRole* ont été envoyés ».

Listing 1: Une exigence d'initialisation du système *S_CP*

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 [5, 12, 10] ont proposé de formuler les propriétés à l'aide de patrons de définition. Le but est de proposer un mode d'expression plus proche des langages que les ingénieurs ont l'habitude de manipuler. Nous reprenons cette approche en réutilisant les patrons de Dwyer et Smith. Nous les étendons et nous les implantons dans le langage de description de contextes CDL.

Une autre difficulté à contourner est de gérer la complexité des preuves due à l'explosion du nombre de comportements du modèle à traiter lors de la vérification. Nous faisons alors un premier constat. Nous avons vu précédemment que dans les documents d'exigences d'un système, celles-ci sont souvent exprimées, informellement, dans un contexte donné de l'exécution du système. Les exigences sont associées à des phases d'utilisation spécifique du système (initialisation, reconfiguration, modes dégradés, changement d'état, etc.). Il n'est donc pas nécessaire de les vérifier sur tous les scénarios de l'environnement. Dans les travaux décrits dans [5, 10], 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*). Ceux-ci permettent de localiser les exigences à vérifier dans un contexte temporel

³ Ces résultats ont été présentés [23] en octobre 2009 lors de la 12^{ème} conférence IEEE/ACM MODELS'09.

particulier d'exécution du modèle à valider. 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'événements. Mais en pratique, dans des contextes d'exécution complexes lorsque, par exemple, l'environnement d'un système est composé de plusieurs acteurs s'exécutant en parallèle, ces opérateurs ne sont pas aisés à utiliser pour spécifier cette localisation. Le déroulement des phases d'exécution de l'environnement peut être alors difficile à décrire avec les opérateurs proposés à cause de l'enchaînement des interactions entre l'environnement et le modèle. Nous proposons donc de mettre en œuvre un lien entre chaque exigence ou propriété à vérifier à une région spécifique d'exécution du modèle à valider. Les propriétés sont localisées explicitement dans un contexte temporel déterminé par le cahier des charges.

Dans le cas de documents d'exigences industriels que nous avons eu à traiter, ce lien exigence-contexte est rarement explicité formellement, parfois transparait dans des descriptions disséminées dans plusieurs documents. Nous proposons donc de lier explicitement et formellement les propriétés formalisées à leur contexte d'exécution spécifiques pour limiter la portée de la propriété. L'avantage est de spécifier explicitement les conditions, c'est à dire, le contexte d'exécution, pour lesquelles une propriété donnée doit être vérifiée. Ceci a pour conséquence de simplifier l'expression de la propriété et d'être facilement plus compréhensible. Ce qui nous amène à un deuxième constat. Pour diminuer l'explosion combinatoire, un moyen peut être de partitionner l'activité de vérification en un ensemble d'actions de vérification. Chaque vérification d'un ensemble de propriétés est exécutée en restreignant le nombre comportement du modèle en le plongeant explicitement dans un contexte spécifique qui se synchronise avec le modèle. Lors d'une preuve, le nombre d'états du système pour lesquels les propriétés sont à vérifier est alors considérablement réduit. Pour que cette approche soit fondée, le processus de développement du système doit inclure une étape de spécification de l'environnement permettant d'identifier explicitement des ensembles de comportements limités mais de manière complète. L'hypothèse forte que nous faisons pour mettre en œuvre ce processus méthodologique est que le concepteur est capable d'identifier toutes les interactions possibles entre le système et son environnement. Nous justifions cette hypothèse, en particulier dans le domaine de l'embarqué, par le fait que le concepteur d'un système doit connaître précisément et complètement le périmètre (contraintes, conditions) de son utilisation pour pouvoir le développer correctement.

Nous mettons en œuvre cette approche en proposant le formalisme CDL comme un langage prototype qui permet à l'ingénieur de décrire les contextes et les propriétés qu'il souhaite vérifier sur son modèle. Nous présentons maintenant, dans les grandes lignes, ce langage.

3 Le langage CDL

Un modèle CDL permet à l'utilisateur de décrire le comportement de l'environnement du modèle à valider et les propriétés devant être vérifiées. Ce DSL basé sur UML2, permet, d'une part, de décrire plusieurs entités contribuant à l'environnement et pouvant s'exécuter en parallèle, comme les acteurs *Device* et *HMI* dans les figures 1 et 2. D'autre part, il intègre un langage de description de propriétés reposant sur la notion de patron. Un méta modèle de CDL a été défini et une sémantique décrite [22] en terme de traces s'inspirant des travaux de [7, 13].

3.1 Description hiérarchique du contexte

Un modèle *CDL* est structuré, de manière hiérarchique, en 3 niveaux (figure 2). 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 enchaînements 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 enchaînements 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.

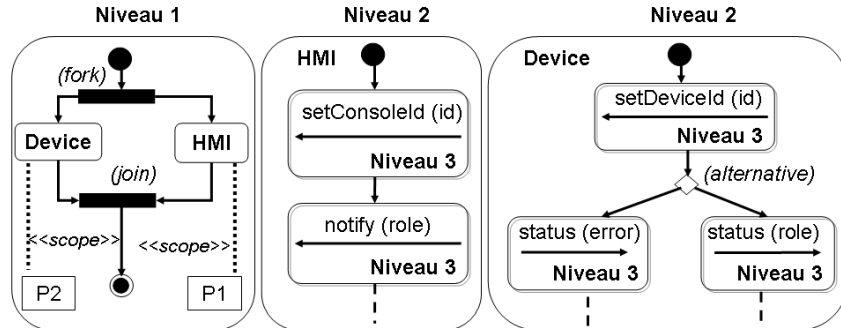


Figure 2 : Le système S_CP : modèle partiel du contexte. Le cas d'utilisation et le diagramme de séquence de la figure 1 sont transformés et complétés pour créer le modèle de contexte. Tous les scénarios du contexte sont spécifiés en CDL à l'aide des opérateurs d'alternative et parallèle.

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 [7] et conformément à la sémantique des diagrammes de séquence d'UML2.0. 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'événements et décrit un historique de l'interaction entre objets, comme par exemple la trace $\langle x!, x? \rangle$ 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 les entités (acteurs) inclus dans le contexte et, de manière hiérarchique, les enchaînements d'activités des entités. 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 et de variables 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 [11], un dépliement 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 séquençement dans les diagrammes d'activités.

3.2 Liaison contexte - propriétés

En plus de ces trois niveaux, un modèle CDL intègre la spécification des propriétés à vérifier. Les propriétés spécifiées sont référencées dans le modèle par des liens stéréotypés *scope*. Une propriété peut être aussi liée à un nœud dans un diagramme de niveaux 1 ou 2. Elle sera à vérifier pour les exécutions associées à ce nœud. Dans une approche de vérification par observateurs, chaque spécifiée est traduite dans l'outillage en un automate observateur. Celui-ci, encodant la 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.

3.3 Spécification des propriétés

Des patrons de définition capturent, sous forme textuelle, des types de propriétés usuellement rencontrées dans les documents d'exigences. Chez [5, 10], 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 événements détectables [22] 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 [10]. Dans CDL, 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 [9]. Les opérateurs *AN* et *ALL* précisent respectivement si un évènement ou tous les évènements, ordonnés (*Order*) ou non (*Combined*), d'un ensemble sont concernés par la propriété.

Nous illustrons dans le listing 2, pour notre cas d'étude *S_CP*, une propriété *PI* qui correspond à une exigence *RI* provenant de la décomposition d'exigence du listing 1 comme expliqué au paragraphe 4 :

Exigence RI : « Au cours de la procédure d'initialisation, *S_CP* doit associer un identificateur aux *NC* consoles (*IHM*), avant un délai *dMax_cons*. »

RI référence la communication entre *S_CP* et les consoles (*IHM*). Dans le diagramme de séquence de la figure 1, l'association à d'autres périphériques n'a aucun effet sur *RI*. Pour l'étude de cas, le nombre de consoles (*HMI*) considéré ici est deux ($NC = 2$). *RI* décrit une observation des occurrences d'évènements. *S_CP_hasReachState_Init* fait référence à un changement d'état d'un processus du modèle à valider. *sendSetConsoleIdToHMI1* et *sendSetConsoleIdToHMI2* fait référence à des envois de signaux décrits dans le modèle CDL (figure 2). Tel que mentionné dans la section 4, notre outil OBP⁴ [11, 21] transforme chaque propriété en un automate observateur [6], incluant un nœud de rejet *reject*. Avec les observateurs, nous pouvons traiter des propriétés de type

⁴ OBP (*Observer-Based Prover*) est disponible sous licence EPL sur le site TopCased : <http://gforge.enseeiht.fr/projects/obp>

sûreté et vivacité bornée. Une analyse d'accessibilité consiste en la recherche d'états *reject* qui sont atteints par un observateur. Dans notre exemple, le nœud *reject* de l'observateur est atteint après la détection de l'événement *S_CP_hasReachState_Init* si la séquence *sendSetConsoleIdToHMI1* et *sendSetConsoleIdToHMI2* n'est pas produite dans l'ordre et avant *dMax_cons* unités de temps. A l'inverse, le nœud de *reject* n'est pas atteint si l'événement *S_CP_hasReachState_Init* n'est jamais reçu, ou si la séquence des deux événements ci-dessus est produite correctement (dans le bon ordre et avec le délai spécifié). Par conséquent, une telle propriété peut être vérifiée à l'aide d'une analyse de l'accessibilité mise en œuvre dans un vérificateur de modèle.

Property P1 ;

```

exactly one occurrence of S_CP_hasReachState_Init
eventually leads-to [0..dMax_cons]
ALL Ordered
    exactly one occurrence of sendSetConsoleIdToHMI1
    exactly one occurrence of sendSetConsoleIdToHMI2
end
S_CP_hasReachState_Init may never occurs
one of sendSetConsoleIdToHMI1 cannot occur before S_CP_hasReachState_Init
one of sendSetConsoleIdToHMI2 cannot occur before S_CP_hasReachState_Init
repeatability : true

```

Listing 2 : Le système S_CP : la propriété de type de réponse correspondant à l'exigence R1.

4 Méthodologie et outillage OBP

4.1 Le processus de vérification

Le processus méthodologique de preuve de modèles que nous proposons est basé sur une activité de vérification d'exigences sur un modèle de conception. Les vérifications ont pour but de contrôler que le modèle conçu est conforme à ses spécifications, c'est à dire à un ensemble d'exigences. Pour établir la vérification de l'ensemble des exigences, nous supposons que celles-ci puissent être formalisées sous la forme de propriétés logiques (comportementales de nature fonctionnelle ou non fonctionnelle). Nous supposons également que l'environnement du modèle (le contexte) à valider ainsi que les interactions entre l'environnement et le modèle soient modélisées formellement. Enfin, en vue d'utiliser un vérificateur formel, le modèle de conception doit pouvoir être simulable. Les propriétés, l'environnement et le modèle simulable constituent les données pertinentes et suffisantes pour conduire les preuves d'exigences sur le modèle.

Nous identifions un processus de vérification d'exigences outillé (figure 3). Ce processus inclut les phases suivantes :

- A partir d'un modèle de conception fourni par l'industriel, un modèle de conception formel est généré (manuellement ou semi-automatiquement) par une extraction des données utiles du modèle de conception. Ces données permettent d'obtenir un modèle comportementale formel simulable.

- A partir d'exigences fournies par le document d'exigences, des propriétés sont formalisées et un ou des modèles du comportement de l'environnement sont construits. Cette formalisation est réalisée à partir d'une interprétation d'exigences textuelles et d'une description de l'environnement du système modélisé. La rédaction d'exigences formelles et la modélisation du contexte sont d'autant plus aisée que, d'une part, la description du comportement de l'environnement est formalisé (cas d'utilisation, scénarios, diagramme de séquence, etc.) et que, d'autre part, les exigences sont exprimées sans ambiguïtés. Dans notre approche, l'utilisateur rédige ses exigences avec les patrons de propriétés proposés dans CDL. Dans l'état actuel du langage CDL, certaines exigences du document d'exigences peuvent ne pas correspondre pas aux patrons proposés.

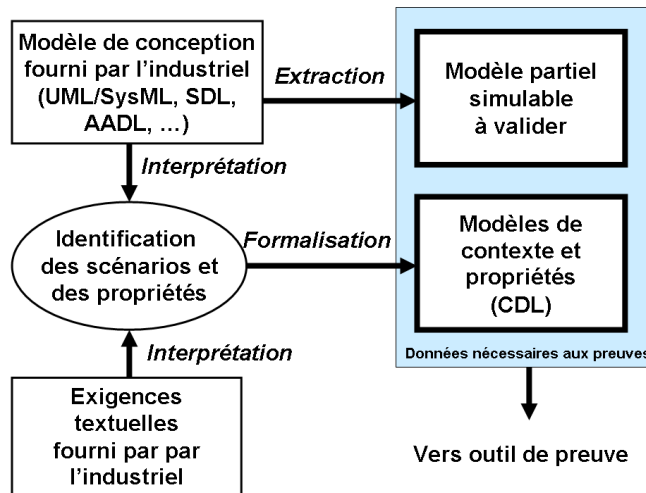


Figure 3 : Processus de prise en compte des données et des modèles du concepteur.

A ce niveau, il nous semble important que la formalisation des exigences et du contexte soit bien dissociée des éléments du modèle de conception et formalisés de manière complète et non ambiguë. Ils doivent également porter, de manière non ambiguë, sur des éléments bien identifiés du modèle de conception et à un même niveau d'abstraction. Le modèle quant à lui doit pouvoir être interprété de manière unique et être exécutable par un ordinateur, moyennant une transformation (sémantique) de modèle adéquat. Dans la définition du processus de vérification, nous avons identifié le concept d'*Unité de Preuve (UdP)* [20] comme une structure de données regroupant toutes les données nécessaires et suffisantes pour conduire la vérification d'un ensemble de propriétés sur un modèle dans un contexte donné. Une *UdP* assure le lien entre le modèle CDL et le modèle à valider (figure 4). A partir des données contenues dans une *UdP*, les codes formels assimilables par les outils de vérification sont générés automatiquement.

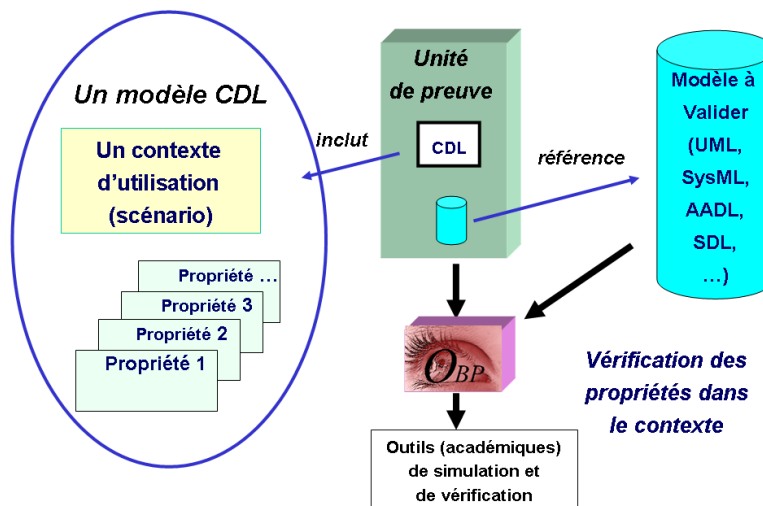


Figure 4 : Modèle CDL et Unité de Preuve

Une condition préalable pour la mise en œuvre de ce processus est de disposer, de la part de l'industriel, de spécifications qui permettent d'identifier les contextes et les exigences qui peuvent être soumises à la vérification et qui peuvent être reliées à un contexte. Dans beaucoup de documents industriels que nous avons traités, les informations portant sur les contextes étaient très souvent implicites ou réparties dans plusieurs documents. Des discussions ont donc été nécessaires avec les ingénieurs pour comprendre les différents contextes du système et les capturer dans les modèles CDL. Le processus de développement doit inclure une étape de spécification de l'environnement permettant d'identifier un ensemble complet de toutes les interactions entre l'environnement et le

modèle, ce qui doit assurer un taux de couverture de 100%. L'atteinte de cette couverture correspond à l'hypothèse formulée précédemment, qui postule que le concepteur est capable d'identifier toutes les interactions possibles entre le système et son environnement. L'ensemble des interactions doit lui être fourni formellement comme un résultat du processus d'analyse de l'architecture logicielle conçue, et ceci dans un processus de développement encadré et outillé. Compte tenu de la complexité de l'ensemble des interactions, il est préférable que l'utilisateur construise un ensemble structuré de modèles CDL spécifiques, chacun correspondant à des cas d'utilisation spécifiques, ce qui implique l'exploitation d'un ensemble d'unités de preuve.

En ce qui concerne la formalisation des propriétés, de nombreuses exigences comportaient beaucoup d'informations et ont du être décomposées en exigences élémentaires pour simplifier leur formalisation avec les patrons. Par exemple, dans l'étude de cas décrit précédemment, l'exigence *R1* (listing 1) peut être décomposée en quatre sous-exigences comme suit :

R1 : « Au cours de la procédure d'initialisation, *S_CP* doit associer un identificateur aux *NC* consoles (*IHM*), avant un délai *dMax_cons.* »
R2 : « Ensuite, *S_CP* doit associer un identificateur aux *NE* périphérique (*Device*), avant un délai *dMax_dev.* »
R3 : « Chaque périphérique renvoie un message *statusRole* à *S_CP* avant un délai *dMax_ack.* »
R4 : « *S_CP* transmet un message *notifyRole* à chaque périphérique et chaque console connectés. L'initialisation s'achève avec succès, lorsque *S_CP* a affecté tous les identificateurs de périphérique et de console et lorsque tous les messages *notifyRole* ont été envoyés. »

Une fois ce travail de décomposition des exigences réalisé, la formalisation avec les patrons CDL est beaucoup plus aisée.

4.2 Le prototype OBP (*Observer-Based Prover*)

Les modèles CDL sont traduits, dans l'outil OBP (figure 5), en codes assimilables par un vérificateur. Dans nos expérimentations, OBP génère des programmes basés sur des automates temporisés [1] au format IF2 [2] en vue d'alimenter le simulateur IFx [2]. Des développements sont cours pour générer des programmes FIACRE [15] et se connecter aux outils TINA [14] ou CADP [8]. IFx génère un système de transitions qui est analysé par OBP pour délivrer à l'utilisateur un résultat de preuve compréhensible (état de véracité des exigences et contre exemples filtrés).

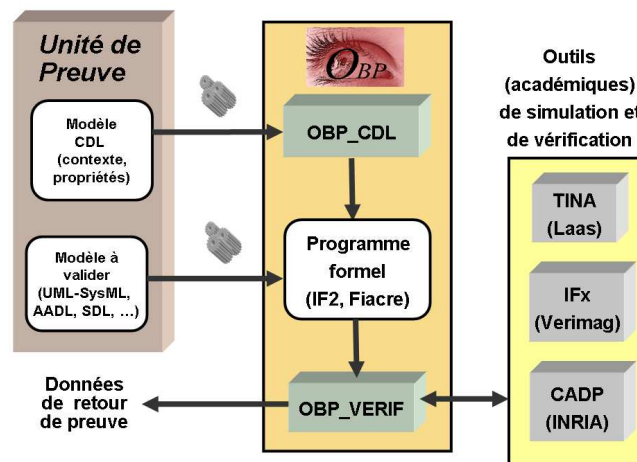


Figure 5 : L'outil OBP (*Observer-Based Prover*)

L'utilisation des outils de vérification connectés à l'outil OBP implique des transformations des modèles d'entrée de l'utilisateur et des modèles de contexte CDL vers les programmes formels exploités par les outils de vérification. Des chaînes de transformation sont implantées dans OBP. Actuellement, les modèles de conception à valider sont importés dans OBP (figure 5) au format IF2. Pour importer des modèles dans des formats UML 2, AADL [19] ou SDL [4], il est nécessaire

de mettre en œuvre des traducteurs adéquats comme ceux étudiés dans les projets TopCased⁵ ou Oméga⁶. Ils permettront de générer automatiquement, à partir des modèles de conception de l'utilisateur des programmes formels IF2 ou FIACRE.

Les modèles CDL sont édités dans un environnement Eclipse et importés dans OBP au format XMI⁷. OBP les interprète et génère des automates dans le format intermédiaire des *α -contextes* proposé dans [11] que nous avons étendu pour prendre en compte les exécutions parallèles des entités de l'environnement, la gestion des variables et des contraintes temporelles. Ces automates sont ensuite composés, dépliés et partitionnés pour produire des automates temporisés linéaires IF2 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, par une technique de transformation de modèles, les automates observateurs à partir des propriétés qui sont décrites dans le modèle CDL.

Les *α -contextes* sont destinés à être raffinés ou *dépliés* en des automates temporisés. La traduction des *α -contextes* implique plusieurs phases automatisées et implantées par des programmes de transformation de modèles écrits en java :

- le dépliage des *α -contextes* en des automates nommés *contextes concrets*. Le dépliage prend 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.

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

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

L'outil OBP a été utilisé dans plusieurs études de cas qui sont décrites au paragraphe suivant.

5 Expérimentations et des résultats

Nous relatons ici l'application de notre approche à six systèmes embarqués dans les domaines de l'avionique embarquée de deux partenaires industriels (dénotés A and B). Pour les cas d'étude (CS₁ à CS₆), quatre des composants logiciels proviennent du partenaire A et les deux derniers du partenaire B. Pour chaque composant, le partenaire industriel a fourni les documents d'exigences opérationnelles (use cases, propriétés exprimées en langage naturel) ainsi qu'un modèle à valider et exécutable. Les modèles exécutables de composant sont décrits soit en UML2, complété par des programmes ADA ou JAVA, ou soit en langage SDL. Le tableau 1 donne le nombre d'exigences

⁵ <http://www.topcased.org>

⁶ <http://www-Omega.imag.fr>

⁷ XML (*eXtensible Markup Language*) *Metadata Interchange*

pour chaque composant, et indique la complexité du composant en termes de lignes de code. Nous appliquons et suivons les étapes de la méthode décrite au paragraphe 4 : spécification des propriétés, description des contextes et construction des unités de preuve.

Tableau 1. Classification des cas d'étude.

	CS ₁	CS ₂	CS ₃	CS ₄	CS ₅ ⁸	CS ₆
Langage de modélisation	SDL	SDL	SDL	SDL	UML2	UML2
Nombre de lignes de code	4 000	15 000	30 000	15 000	38 000 ⁹	25 000 ¹⁰
Nombre de d'exigences	49	94	136	85	188	151

5.1 Spécification des propriétés

Dans cette étape, les exigences décrites en langage naturel sont exprimées sous la forme de propriétés temporelles. Pour créer les modèles CDL en exploitant les patrons de propriétés, nous avons analysé les documents d'exigences textuelles en identifiant les exigences qui pouvaient être traduites dans des automates observateurs. Lors de cette étape, nous avons constaté que la plupart des exigences se déclinaient non pas en une mais en plusieurs propriétés (comme nous l'avons déjà montré dans l'exemple de S_CP). De plus, dans les exigences textuelles, différents niveaux de description, d'abstraction, sont mêlés. Nous avons donc extrait seulement les exigences correspondantes au niveau d'abstraction du modèle à valider. Finalement, nous avons constaté que beaucoup d'exigences textuelles étaient ambiguës. Nous avons donc du lever ces ambiguïtés, et pour cela clarifier la sémantique des exigences par un dialogue avec nos partenaires industriels.

Le tableau 2 indique le nombre de propriétés qui ont pu être traduites depuis les exigences. Nous avons classé les exigences qui ont été traitées en trois catégories en indiquant, pour chaque cas et pour chaque classe, les pourcentages par rapport à l'ensemble des exigences : les exigences « *prouvables* » sont celles qui peuvent être capturées avec notre approche et peuvent être converties en observateurs. La technique de preuve peut être appliquée sur un contexte donné sans explosion combinatoire. La catégorie « *non calculable* » représente les exigences qui peuvent correspondre à un patron, mais ne peuvent pas se traduire par un observateur. Par exemple, les propriétés de vivacité non bornées (*liveness*) ne peuvent être traduites sous forme d'un observateur, qui ne permet que de capturer les propriétés de vivacité bornée. Nous pourrions cependant traiter ces exigences en générant une formule de logique temporelle qui pourrait alimenter un vérificateur comme TINA. La catégorie *non prouvable* correspond aux exigences qui ne peuvent pas du tout être interprétées avec notre approche. C'est le cas lorsqu'une propriété fait référence à des événements indétectables (non observables) pour un observateur, tels que l'absence d'un signal.

Tableau 2. Nombre de propriétés traitées.

	CS ₁	CS ₂	CS ₃	CS ₄	CS ₅	CS ₆	Moyenne
Prouvables	38/49 (78%)	73/94 (78%)	72/136 (53%)	49/85 (58%)	155/188 (82%)	41/151 (27%)	428/703 (61%)
Non calculables	0/49 (0%)	2/94 (2%)	24/136 (18%)	2/85 (2%)	18/188 (10%)	48/151 (32%)	94/703 (13%)
Non Prouvables	11/49 (22%)	19/94 (20%)	40/136 (29%)	34/85 (40%)	15/188 (8%)	62/151 (41%)	181/703 (26%)

Pour le cas CS₆, nous constatons que le pourcentage d'exigences prouvables (27 %) est faible. En effet, il était très difficile de ré-exprimer les exigences depuis la documentation dans le cadre des patrons proposés. Nous aurions du consacrer beaucoup plus de temps pour interpréter les exigences

⁸ CS₅ correspond à l'étude de cas S_CP décrit partiellement dans la section 2

⁹ Le modèle UML est implémenté par un programme de 38 000 lignes de code ADA.

¹⁰ Le modèle UML est implémenté par un programme de 25 000 lignes de code JAVA.

avec notre partenaire industriel pour les aligner avec nos patrons. Par contre, pour le CS₅, nous notons que le pourcentage (82 %) des propriétés prouvables est très élevé. En effet, la majorité des 188 exigences étaient en adéquation avec les patrons. Ceci est dû au fait que nous avons eu des discussions très tôt dans le processus de développement avec les ingénieurs en charge de rédiger les exigences, ce qui a permis d'influer sur leur rédaction initiale.

5.2 Description du contexte des propriétés

Après que les propriétés aient été exprimées à l'aide de nos patrons, nous avons lié chaque propriété à des scénarios de l'environnement. Ici, le travail a consisté à transformer les cas d'utilisation en contexte avec notre langage CDL. Un ou plusieurs contextes CDL ont été créés en fonction de la complexité des contextes de comportement et du nombre d'acteurs interagissant avec l'environnement. Le tableau 3 indique le nombre de chemins obtenu pour différents modèles CDL pour l'étude du cas CS₁. Ce nombre dépend des opérateurs *alternative* et *parallel*, des acteurs du système et des interactions utilisées dans le modèle CDL. A chaque modèle, est attaché un ensemble de propriétés correspondant à une phase spécifique ou à des scénarios. Nous constatons que le temps de vérification peut être long (par exemple, pour le cas CS₁, temps de 20 minutes pour CDL₄) parce que le temps de compilation de l'état d'un graphe IFx est répété pour chaque chemin de contexte. Il s'agit d'un point d'optimisation possible, qui consisterait à limiter le partitionnement des chemins ou à calculer des équivalences des chemins pour une propriété particulière.

Tableau 3. Nombre de modèles CDL et de chemins générés pour le cas CS₁.

	CDL ₁	CDL ₂	CDL ₃	CDL ₄	CDL ₅
Nombre d'acteurs	1	3	3	5	3
Nombre de chemins	3	128	82	612	96
Temps de vérification (sec.)	6	256	164	1224	192

5.3 Exploitation des unités de preuve

Dans les études de cas, pour chaque modèle CDL, une unité de preuve est créée. Une unité de preuve permet de combiner un ensemble d'observateurs en fonction d'un contexte. Pour chaque chemin généré par OBP, un graphe d'accessibilité est généré et représente l'ensemble de toutes les exécutions possibles du modèle. Une propriété est dite « non vérifiée » par l'outil si un état "reject" d'un automate observateur correspondant est atteint : OBP affiche la liste des observateurs en erreur et les contre exemples. L'explosion combinatoire peut se produire lors de la simulation. La création des contextes spécifiques adéquats permet de limiter le nombre des comportements pour un modèle. L'identification des configurations spécifiques qui limitent les espaces d'exploration du modèle doit être opérée avec méthode, comme évoqué précédemment pour que la preuve des propriétés reste fondée.

6 Discussion et conclusion

Le prototype OBP et le langage CDL ont pu être utilisés pour mener des expérimentations, en contexte industriel. Pour chaque étude de cas, ces vérifications ont été réalisées par les ingénieurs eux-mêmes, avec l'aide de notre équipe, dans les limites des fonctionnalités offertes par l'outil et du niveau d'expression actuel du langage CDL. Cette technique a permis aux utilisateurs d'obtenir un diagnostic pour chaque exigence traitée.

6.1 Bénéfices de l'approche

La formalisation du comportement de l'environnement (contextes). L'apport des modèles CDL, en relation avec le modèle à valider, est d'offrir, d'une part, un cadre pour décrire le comportement des acteurs de l'environnement du modèle. D'autre part, la description de l'environnement permet de restreindre l'exécution du modèle qui interagit avec lui. Parfois, dans

des applications industrielles, cette description est informelle et/ou non complète. L'approche modèle permet à l'utilisateur de formaliser cet environnement et de préciser, dans un ensemble de modèles CDL, les cas d'utilisation du composant développé. Cette formalisation ne peut être que bénéfique pour une meilleure conception, même si l'utilisateur ne l'exploite pas, par la suite, pour des analyses formelles. Pour certains modèles, le nombre de modèles CDL peut être important selon la manière dont le système interagit avec les acteurs de l'environnement. C'est suite à une analyse des fonctionnalités du système à développer que l'utilisateur peut décrire exhaustivement l'environnement et les modéliser en CDL. Ceux-ci sont basés sur des diagrammes d'activité « à la UML » et des diagrammes de séquences qui sont d'un accès aisé pour un ingénieur. Conceptuellement, les principes de CDL peuvent être implantés dans d'autres formalismes.

La synthèse des résultats de preuves, montrée précédemment, fait apparaître la possibilité d'exploiter des ensembles de contextes pour réduire la complexité des modèles à valider. Pour chaque étude de cas, il a été possible de créer des unités de preuve, incluant chacune un modèle CDL, et les fournir en entrée d'OBP pour la génération des ensembles de chemins (automates linéaires). Chaque chemin est composé avec le modèle du composant à valider et les automates observateurs. C'est cette composition qui permet de réduire la taille (finie) du système de transition généré lors de la simulation exhaustive et d'arriver à un résultat de preuve en un temps raisonnable pour l'utilisateur. Dans ce cas, CDL contribue à surmonter l'explosion combinatoire en rendant possible une vérification partielle sur un ensemble restreint de scénarios spécifiés par les automates de contexte.

La formalisation des exigences. Les résultats montrent également qu'une grande partie des exigences, devant être traitées, a pu faire l'objet d'une formalisation basée sur les patrons de définition proposés et ont pu donner lieu à la génération d'automates observateurs. L'utilisateur peut décrire les exigences, sous une forme textuelle encadrée et non ambiguë, conforme à une grammaire. Celle-ci est suffisamment expressive pour décrire, sans trop de difficulté pour l'utilisateur, les propriétés de réponse, d'existence et d'absence qui sont des propriétés de sûreté ou de vivacité bornée. Celles-ci sont bien adaptées pour décrire des comportements caractérisant les interactions entre entités ou composants communicants. Elles correspondent, dans ce domaine, à un grand pourcentage des exigences exprimées. Ce mode d'expression permet de lever toute erreur d'interprétation car la sémantique des patrons est clairement définie (sémantique de la logique linéaire [18]). Chaque exigence formalisée peut être ensuite traduite automatiquement en un programme formel (automate observateur) qui se compose, lors de la vérification, avec le programme associé au modèle. Il faut noter que dans le domaine des protocoles, ce type de propriété est majoritaire.

Les résultats de preuve. Lors de l'étude de l'ensemble des exigences pour chaque cas d'étude, en moyenne 13 % de celles-ci (non-calculable) (cf. tableau 2) correspondait à des propriétés de vivacité non traduisibles, avec notre approche, en observateurs. Elles ont donc demandé, après discussion avec les utilisateurs, une adaptation pour les mettre sous la forme de propriétés de vivacité bornée. Les 61 % (prouvables) ont été réécrites sans difficultés avec les patrons. Pour le reste des 26 % (non prouvables), elles doivent être encore discutées avec les partenaires industriels pour trouver une expression équivalente et prouvable. Dans les études de cas, environ quarante exigences importantes ont été formellement vérifiées avec nos outils. Lors des preuves, nous avons constaté, dans deux études de cas (CS₁ et CS₅), une exécution qui n'a pas satisfait aux exigences. Ces deux cas étude correspondent à des systèmes embarqués déjà actuellement opérationnels. Les techniques classiques de simulation n'avaient pas, jusque là, permis de trouver ces erreurs. Une des raisons est que ces erreurs apparaissaient dans des contextes spécifiques ne correspondant pas à des scénarios réellement opérationnels. Ceux-ci n'avaient donc pas été testés lors des simulations antérieures, ni lors des tests sur cible. Mais, aucun document de spécification des deux systèmes ne le précisait explicitement. Ceci montre qu'un manque de formalisation explicite des cas d'utilisation peut laisser un flou dans la spécification lors la conception.

6.2 L'appropriation du langage CDL

Dans le cas d'étude, les diagrammes de contexte furent construits, d'une part, à partir de scénarios décrits dans les documents de conception et, d'autre part, à partir des documents d'exigence fournis par les industriels. Deux difficultés majeures sont alors soulevées. La première est le manque de description complète et cohérente du comportement de l'environnement. Les cas d'utilisation décrivant les interactions entre un système (S_CP par exemple) et son environnement sont souvent incomplets. Par exemple, des données concernant les modes d'interaction peuvent être implicites. Le développement des diagrammes CDL requiert alors beaucoup de discussions avec des experts qui ont conçu les modèles afin d'explicitier toutes les hypothèses associées au contexte.

La deuxième difficulté concerne la compréhension des exigences. Le problème provient de la difficulté à les formaliser en propriétés formelles. Celles-ci sont regroupées dans des documents de niveaux d'abstraction différents correspondant 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 qui en ont une connaissance parfaite.

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. CDL invoque un style de spécification très proche d'UML et donc lisible par les ingénieurs. Dans toutes les études de cas, les collaborateurs industriels ont indiqué que les modèles CDL améliorent la communication entre les développeurs possédant différents niveaux d'expérience. Ils les guident dans leur travail de structuration et de formalisation de la description des environnements de leurs systèmes et de leurs exigences. La collaboration avec les ingénieurs responsables de la rédaction des exigences les a motivés pour une prise en compte d'une approche plus formelle, ce qui est une amélioration par rapport à leurs pratiques précédentes. Nous pensons donc que de telles approches contribuent à une meilleure appropriation industrielle des techniques de vérification formelle.

Les contextes et les propriétés sont des artefacts qu'il faut créer dans le but d'appliquer des techniques de vérification. Ce sont les données qui sont rassemblées dans des unités de preuve en vue d'effectuer des activités de preuve pour valider des modèles. Elles doivent être capitalisées, au sein des unités de preuve, si les modèles de conception évoluent au cours du cycle de développement. Il semble donc essentiel d'étudier un cadre méthodologique pour aider l'utilisateur à décrire et formaliser les contextes de preuve en tant que composants répertoriés dans un processus déroulant l'activité de vérification.

6.3 Travaux en perspective

Aujourd'hui, l'approche a été mise en œuvre dans un nombre limité de cas industriels (une dizaine de cas chez divers partenaires). Mais les résultats obtenus dans les expérimentations avec le langage CDL et OBP sont très encourageants et ils ouvrent, bien sûr, tout un ensemble d'axes de travaux de recherche à entreprendre.

Dans le cas de modèles de contexte CDL complexes (acteurs multiples, nombreuses alternatives dans les comportements des acteurs), la génération par OBP de l'ensemble des chemins peut mener à un grand nombre de chemins. Cela multiplie les temps de composition, de simulation et

d'analyse d'accessibilité des états d'erreur. En effet, la complexité d'analyse du modèle a disparue du fait de la technique de restriction des comportements du modèle par composition avec des contextes partiels (chemins). Mais cette complexité a été déportée, en amont, lors de la génération des chemins à exploiter. Face à ce problème, un travail est en cours [24] pour réduire ces ensembles. Nous cherchons à limiter le nombre de chemins pertinents en identifiant des critères d'équivalence qui permettent d'éliminer des chemins équivalents au regard des propriétés à vérifier.

L'affichage des retours de preuve provenant des model-checkers pose également un problème de compréhension pour l'utilisateur. Aujourd'hui OBP affiche les chemins pour lesquels des observateurs ont été falsifiés. Mais ces chemins peuvent être de très grande taille (plusieurs milliers de transitions) et sont donc difficilement interprétables par l'utilisateur car ils décrivent des exécutions à un niveau sémantique très bas. Il faut donc concevoir des filtres qui n'affichent que des informations pertinentes et en nombre réduit et les animer dans les modèles de l'utilisateur. Une étude est en cours pour l'implantation d'interpréteurs des données de retours de preuve prenant en compte les intentions de vérification (contexte et exigences) et les relations entre les entités de l'environnement et celles du modèle.

Suite à la mise en œuvre de l'approche et des expérimentations, nous avons la nécessité de formaliser la méthodologie. La manière d'aborder la construction des modèles CDL, inclus dans les unités de preuves, a une implication directe sur la complexité engendrée lors de la preuve. Nous cherchons donc à pouvoir aiguiller l'utilisateur dans sa construction des modèles CDL. Les modèles manipulés par l'utilisateur en phase d'analyse d'un système (cas d'utilisation, diagrammes de séquences, etc.) doivent pouvoir être exploités pour générer automatiquement les modèles CDL et donc les unités de preuve. Ceci doit s'effectuer, d'une manière complète, pour obtenir une couverture complète des cas d'utilisation. Les diagrammes de contexte CDL doivent aussi pouvoir être exploités pour concevoir les tests de l'implantation finale puisqu'ils modélisent les interactions avec le composant à valider.

Références

1. Alur R, Dill D.: A Theory of Timed Automata. In Theoretical computer Science, 126(2), pp. 183-235 (2004)
2. Bozga M., Graf S., Mounier L.: IF2: A validation environment for component-based real-time systems. In Proceedings of Conference on Computer Aided Verification, CAV'02, Copenhagen, LNCS. Springer Verlag (2002)
3. Clarke T., Evans A., Sammut P., Willians J.: Applied Meamodeling: A foundation for Language Driven Development. Technical report, version 0.1, Xactium (2004)
4. ITU-T. Recommendations Z-100. Specification and Description Language (SDL) (1994)
5. Dwyer M.B., Avrunin G.S., Corbett J.C.: Patterns in property specifications for finite-state verification. In Proc. of the 21st Int. Conf. on Software Engineering, pp. 411-420. IEEE Computer Society Press (1999)
6. Halbwachs N., Lagnier F., Raymond P.: Synchronous observers and the verification of reactive systems. In 3rd int. Conf. on Algebraic Methodology and Software Technology (AMAST'93) (1993)
7. Haugen O., Husa K.E., Runde R.K., Stolen K.: Stairs: Towards formal design with sequence diagrams. In journal of Software and System Modeling (2005)
8. Fernandez J-C et al.: « CADP: A Protocol Validation and Verification Toolbox », in Alur R. and Henzinger T.A, editors, Proceedings of CAV'96 (new Brunswick, USA), Vol. 1102 LNCS, August (1996)
9. Janssen W., Mateescu R., Mauw S., Fennema P., Stappen P.: Model Checking for Managers. Conference Spin'99, pp. 92-107 (1999)
10. Konrad S., Cheng B.: Real-Time Specification Patterns. In Proc. Of the 27th Int. Conf. on Software Engineering (ICSE05), St Louis, MO, USA (2005)
11. Roger J.C. Exploitation de contextes et d'observateurs pour la vérification formelle de modèles, Phd report, Univ. of Rennes I (2006)
12. Smith R., Avrunin G.S., Clarke L. and Osterweil L.: Propel: An Approach Supporting Property Elucidation. In Proc. of the 24st Int. Conf. on Software Engineering, ACM Press, pp. 11-21 (2002)

13. Whittle J.: Specifying precise use cases with use case charts. In MoDELS'06, Satellite Events, pp. 290–301 (2005)
14. Berthomieu B., Vernadat F.: Time Petri nets analysis with TINA. 3rd Int. Conf. on the Quantitative Evaluation of Systems (QEST'06), Riverside (USA), pp. 123-124 (2006)
15. Berthomieu B, Bodeveix JP., Filali M., Garavel H., Lang F., Peres F., Saad R, Stoecker J., Vernadat F.: The Syntax and Semantics of FIACRE, Version 1.0 alpha. Technical report projet ANR05RNTL03101 OpenEmbeDD (2007)
16. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 2, pp. 244–263 (1986)
17. Hassine, J.; Rilling, J., Dssouli, R. Use Case Maps as a property specification language, Software System Model, 8, pp. 205-220 (2009)
18. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer, New York (1992)
19. Feiler, P., Gluch D.P., Hudak J.J.: The Architecture Analysis and Design Language (AADL):An introduction. Technical report, Society of Automotive Engineers (SAE) (2006)
20. Dhaussy P., Boniol F.: Mise en œuvre de composants MDA pour la validation formelle de modèles de systèmes d'information embarqués. pp. 133–157 RSTI (2007)
21. Dhaussy P., Auvray J., De Belloy S., Boniol F., Landel E.: Using context descriptions and property definition patterns for software formal verification, Workshop Modevva'08 (hosted by ICST 2008), Lillehammer, Norway (2008)
22. Dhaussy P., Creff S., Pillain P.Y., Leilde V.: CDL language specification (Context Description Language). Technical report version N° DTN/2009/8, ENSIETA (2009)
23. Dhaussy P., Pierre-Yves Pillain PY., Creff S., Raji A., Le Traon Y., Baudry B. *Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation*. In Lecture Notes in Computer Science 5795, Springer Verlag, Andy Schuerr, Bran Selic (Eds): 12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (Models'09), No 5795 (2009), pages 438-452.
24. Dumas X., Boniol F., Dhaussy P., Bonnafous E., *Partial Order Application for Software Formal Verification*, Conférence ERTS'10, Toulouse, 2010, 19-21 mai 2010.