Vérification formelle de propriétés basée sur une réduction de l'espace d'exploration de modèles

Luka Le Roux, Philippe Dhaussy et Frédéric Boniol

Résumé : Cet article décrit une technique de vérification formelle de propriétés basée sur la réduction de l'espace d'exploration du modèle à valider. L'approche empruntée repose sur l'identification de contextes et leur exploitation lors de l'exploration des modèles. Le principe est d'amener l'explorateur à concentrer ses efforts non plus sur l'exploration de l'espace complet des comportements, mais sur une restriction pertinente de ce dernier pour la vérification de propriétés spécifiques. La vérification de propriétés devient alors possible sur l'espace d'états ainsi réduit. Les modèles de contextes et les propriétés sont exprimés dans un langage (CDL pour *Context Description Language*) instrumenté à l'aide d'un outil, nommé OBP (*Observer-Based Prover*). Cette approche est illustrée par son application à un modèle simplifié de contrôleur de pacemaker.

Mots clés : Vérification formelle, exploration de modèle, contextes, observateurs, CDL, FIACRE, OBP

1. INTRODUCTION

Dans le domaine de la validation formelle des logiciels embarqués, beaucoup de techniques de vérification de propriétés ont été explorées. Parmi celles-ci, les techniques de *model-checking* [4,19] ont été fortement popularisées grâce à leur faculté d'exécuter automatiquement des preuves de propriétés. De nombreux outils (*model-checkers*) ont été développés dans ce but [2,9,14,15]. Le fonctionnement général de ces outils consiste à modéliser de façon compacte et abstraite l'ensemble des comportements possibles du système à valider et de son environnement. Ce dernier est nécessaire pour « fermer » le modèle du système avec l'ensemble complet des cas d'utilisation de l'environnement avec lequel le modèle est sensé interagir. Le modèle est ensuite parcouru pour pouvoir décider si l'ensemble des exécutions possibles satisfait les propriétés exprimées.

La complexité des logiciels augmentant, elle apparaît de deux types. Premièrement, une complexité externe est liée à l'environnement que le système embarqué doit contrôler. Cet environnement est en effet de plus en plus souvent composé de plusieurs entités physiques distinctes mais dépendantes et qui doivent être contrôlées en parallèle et en cohérence. C'est le cas, par exemple, d'un système des commandes de vol d'un avion qui pilote plus de vingt gouvernes en parallèle et, pour ce faire, est connecté à plus de trente capteurs. Deuxièmement, une complexité interne est liée à l'architecture logicielle du système embarqué. Cette architecture est également de plus en plus souvent composée de plusieurs processus logiciels interagissant en parallèle. Dans les deux cas (interne et externe), le parallélisme d'entités ou de processus conduit à une explosion combinatoire du nombre de comportements possibles.

Lors d'une exploration d'un modèle, la rapidité du parcours dépend du degré de compactage de l'ensemble des comportements. Beaucoup de travaux ont été menés dans ce sens [1,3,7,10,16,17,18,21]. Cependant, compte tenu des très grandes tailles des ensembles considérés, les progrès réels des outils de vérification ne permettent pas encore de traiter des systèmes réels de taille industrielle.

Nous exposons dans cet article une voie complémentaire s'appuyant sur l'exploration partielle de modèles, voie qui permet dans certains cas pertinents de limiter l'explosion combinatoire. Nous développons les principes de cette approche dans le cade d'une modélisation en langage Fiacre [8] des systèmes à valider et de l'exploitation d'un langage, nommé CDL (*Context Description Language*) [6], pour la description des cas d'utilisation de l'environnement considéré pour les validations. Cette approche est instrumentée à l'aide d'un outil, nommé OBP[1] (*Observer-Based Prover*), qui permet d'importer des modèles CDL, d'explorer des modèles au format Fiacre et d'effectuer des vérifications de propriétés par des analyses d'accessibilité. Nous illustrons, dans cet article, l'apport de cette approche sur un cas d'étude : un contrôleur de pacemaker.

L'article est organisé comme suit. Dans le chapitre 2, nous présentons les principes classiques de vérification de propriétés formalisées sous la forme d'observateurs. Nous les illustrons sur le cas d'étude et donnons des résultats de l'exploration du modèle. Au chapitre 3, nous introduisons la technique de réduction de l'espace des états explorés basée sur la notion de contextes et leur partitionnement. Le langage CDL est présenté au

chapitre 4. Nous tirons un bilan de cette approche et concluons cet article en section 5.

2. PRINCIPES DE LA VÉRIFICATION DE PROPRIÉTÉS PAR EXPLORATION DU MODÈLE

Pour établir la vérification d'un ensemble d'exigences sur un modèle, il faut disposer d'un modèle simulable et des exigences formalisées sous la forme, par exemple, de formules logiques ou d'automates observateurs. Ce modèle est ensuite simulé et exploré par un outil de vérification. L'exploration génère un système de transitions (*SdT*). Celui-ci représente tous les comportements du modèle dans son environnement sous la forme d'un graphe de configurations et de transitions. Sur ce *SdT*, une vérification des propriétés peut être conduite, soit en appliquant des algorithmes de *model-checking* sur les formules logiques, soit en appliquant une analyse d'accessibilité des états d'erreur des observateurs. La difficulté liée à cette technique est la production du *SdT* qui peut être de grande taille, dépassant la taille mémoire disponible (explosion combinatoire). Nous illustrons ce fait sur un exemple déduit et simplifié d'un contrôleur de pacemaker.

2.1 L'étude de cas : un pacemaker

Considérons un contrôleur de pacemaker (figure 1) dont l'environnement est composé d'un système externe, nommé *DCM* (*Device Control Monitor*), et du cœur. Le *DCM* a pour fonction d'interagir avec le contrôleur du pacemaker pour lui fournir des paramètres de fonctionnement. Le modèle du pacemaker se compose d'un processus *Controller* qui reçoit les consignes du *DCM* (modes et valeurs des paramètres), d'un processus *Cardio* qui reçoit les signaux de deux processus *LeadA* et de *LeadV* simulant des capteurs stimulés par le cœur. Le processus *Cardio* se charge en retour d'agir sur *LeadA* et *LeadV* en fonction des modes et paramètres reçus du *DCM* via *Controller*[2].

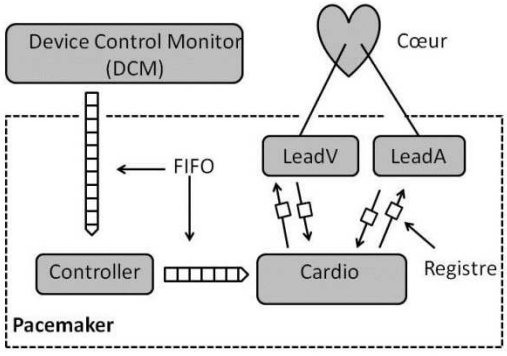

Figure 1 : Modèle d'architecture du pacemaker

2.2 Cas d'utilisation

Le *DCM* interagit avec le pacemaker en lui envoyant des valeurs pour différents paramètres. Chaque paramètre pouvant prendre plusieurs valeurs potentielles que nous avons pris en compte dans notre modélisation :

- paramètre *mode* : 7 modes possibles : VOO, AOO, VVI, AAI, VVT, AAT, VDD.
- paramètre *ARP* : uniquement dans les modes AAI et AAT avec 36 valeurs possibles dans chacun d'eux (de 150 à 500 ms avec un pas de 10 ms).
- paramètre *VRP* : uniquement dans les modes VVI, VVT et VDD avec 36 valeurs possibles dans chacun d'eux (de 150 à 500 ms avec un pas de 10 ms).

paramètre *AVDelay* : uniquement dans le mode VDD avec 24 valeurs possibles (de 70 à 300 ms avec un pas de 10 ms).

- paramètre *LRL* : dans tous les modes, sauf AOO et VOO, avec 62 valeurs possibles (de 30 à 175 ppm, avec un pas de 5 ppm jusqu'à 50 ppm, puis un pas de 1 ppm jusqu'à 90 ppm, puis un pas de 5 ppm jusqu'à 175 ppm).

Dans ce contexte, le *DCM* envoit les valeurs pour cinq paramètres : *mode, ARP, VRP, AVDelay* et *LRL* dans un ordre quelconque. Dans notre modélisation, l'envoi des valeurs par le *DCM* au pacemaker s'effectue par la communication d'un message *chgParam* à destination du processus *Controller*. Ce message a deux arguments : le nom du paramètre à changer et la nouvelle valeur. Par exemple, pour mettre le pacemaker dans le mode *VDD*, le message émis est : *chgParam (mode,VDD).* La figure 2 illustre une interaction mettant en œuvre les cinq messages pour les paramètres mentionnés.

Le nombre de valeurs que peuvent prendre les paramètres dépend donc des modes. Le nombre total de combinaisons possibles, c'est-à-dire d'instances possibles du scénario présenté en figure 2, est ainsi de 62 498.

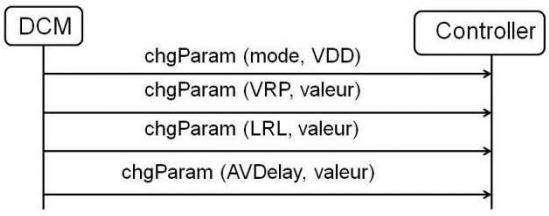

Figure 2 : Un exemple de scénario d'interaction (en mode *VDD*) entre *DCM* et *PGController*

2.3 Exemple de formalisation d'exigence

Pour illustrer les vérifications pouvant être faites sur le modèle, nous nous intéressons à une exigence particulière mais pertinente dans un contexte particulier lorsque le pacemaker est en

mode nominal, c'est-à-dire ambulatoire. Compte tenu des spécifications fournies, les paramètres pertinents à prendre en compte sont : le choix parmi les sept modes (*mode*) et les différentes valeurs *ARP, VRP, AVDelay* et *LRL*.

Nous nous intéressons ici à un exemple d'exigence que nous souhaitons vérifier sur le modèle Fiacre. Celle-ci traite du respect des paramètres spécifiés via le *DCM*. Plus précisément, le paramètre *LowerRateLimit* (*LRL*) indique la période minimale du pouls en deçà de laquelle le cœur doit être stimulé par le pacemaker. Cela signifie qu'un cycle ne doit pas durer au delà de cette période. L'exigence peut s'exprimer informellement ainsi :

Exigence *R1* : *La période d'un cycle ne doit pas être supérieure à la durée LRL.*

L'exigence R1 peut s'exprimer sous la forme d'un invariant sur le modèle. Mais ceci nécessiterait d'instrumenter le modèle par des variables supplémentaires qui pourraient être référencées par l'invariant. Sans ces variables, le temps passé dans le cycle courant n'est pas connu de notre système. Nous choisissons donc une alternative qui consiste à encoder l'exigence par un automate observateur [11] comme illustré figure 3.

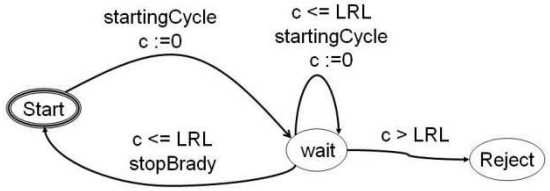

Figure 3 : Observateur temporisé encodant l'exigence R1

Un automate observateur est un automate qui est sensible à des événements survenant lors de l'exploration du modèle et de l'environnement : envois et réceptions de messages, changement d'états des processus et des valeurs de variables. À chaque exécution d'une transition du modèle ou de l'environnement, l'observateur exécute une transition (si elle existe) correspondante à un des événements survenus durant la transition. L'observateur possède un état particulier *Reject*. L'accès à l'état *Reject* signifie que la propriété encodée par l'automate est falsifiée. Une analyse d'accessibilité sur le graphe des états (*SdT*) consiste alors en la recherche de scénarios observés lors de l'exploration du modèle du système et de son environnement conduisant à l'état *Reject* de l'observateur. Plusieurs observateurs peuvent être ainsi définis pour une même exploration du modèle. Avec ceux-ci, nous pouvons ainsi encoder des propriétés de type sûreté et vivacité bornée. L'intérêt du codage par observateurs est de pouvoir exprimer certaines propriétés parfois plus difficilement exprimables par des logiques temporelles telles que *LTL* ou *CTL*.

Concernant l'exigence *R1*, pour être capable de mesurer le temps séparant deux débuts de cycle, nous temporisons l'observateur. Pour cela, nous dotons l'observateur d'une horloge *c* dont la valeur s'incrémente de manière synchrone avec le temps de simulation du modèle. L'horloge *c* peut être remise à zéro lors de l'exécution d'une transition. Les horloges des automates observateurs sont gérées au sein de l'outil OBP conformément à la sémantique des TTS (*Timed Transition Systems*) [13].

Pour modéliser l'observateur figure 3, nous utilisons 2 événements particuliers détectables dans le modèle, *startingCycle* et *stopBrady*. Le premier événement caractérise le début d'un cycle de stimulation et le second est détecté lors de la mise en pause de la stimulation. L'automate de l'observateur passe à l'état *wait* dès l'occurrence de *startingCycle* et retourne, lors de l'occurrence de *stopBrady*, dans son état initial (*Start*) dans l'attente d'un nouveau début de cycle. Le nœud *Reject* de l'observateur est atteint si la valeur de l'horloge *c* dépasse la valeur LRL. L'horloge est remise à zéro à chaque détection de *startingCycle*. La valeur de l'horloge *c* représente ainsi le temps passé dans un même cycle. Nous décrivons au chapitre 4 la formalisation de cet observateur en langage CDL.

2.4 Vérification de l'exigence

Lors de la vérification d'une propriété sur le modèle, celui-ci doit être composé avec les cas d'utilisation décrivant l'environnement du système, puis simulé et exploré par l'outil de vérification. L'environnement représente tous les comportements du modèle dans son environnement sous la forme d'un graphe de configurations et de transitions.

Pour pouvoir vérifier une propriété, il faut donc intégrer, dans le modèle Fiacre, le comportement du processus *DCM* pour simuler l'envoi des valeurs pour les cinq paramètres au processus *Controller*. OBP génère le graphe des comportements en prenant en compte les propriétés à vérifier (invariants et observateurs) et effectue la mise à jour des observateurs en même temps que la génération du graphe des comportements. La vérification, dans le cas des observateurs, se ramène simplement à la recherche des nœuds *Reject* dans le graphe.

Le tableau 1 présente les résultats[3] pour la vérification de l'exigence *R1*, pour tous les modes, avec l'outil OBP. Tous les modes impliquent l'application de l'ensemble des 62 498

combinaisons. L'explosion combinatoire survient à partir de 352 combinaisons (soit moins de 0,6% de l'espace des combinaisons) ce qui nous empêche également de vérifier la propriété au delà de cette complexité.

Nombre de combinaisons	Nombre de Configu-rations explorées	Nombre de Transitions explorées	Temps d'explo-ration (secondes)
7	40 455	48 781	2
42	299 740	364 391	10
110	744 224	909 908	25
226	1 783 438	2 238 305	64
352	Explosion	-	-

Tableau 1 : Vérification avec OBP de l'observateur correspondant à l'exigence *R1*

Pour pouvoir surmonter cette difficulté, et vérifier les propriétés sur un nombre de combinaisons plus important, nous allons mettre en œuvre une technique qui a pour but de limiter la complexité des *SdT* lors de chaque exploration en décomposant l'environnement en plusieurs cas d'utilisation. L'objectif est de vérifier les exigences, non plus lors d'une seule exploration, mais sur un ensemble d'explorations. Nous décrivons cette technique dans le chapitre suivant.

3. APPROCHE DE LA RÉDUCTION

3.1 Principe

Lors de la mise en œuvre classique du *model-checking*, le modèle exploré inclut les scénarios d'utilisation. Ceux-ci décrivent les interactions entre les composants de l'environnement et le modèle. Considérer le modèle avec le comportement de l'environnement amène à devoir explorer un espace d'états la plupart du temps très grand.

Dans notre approche, nous cherchons à traiter les deux sources de la complexité : la complexité externe (c.-à-d. celle de l'environnement) et la complexité interne (c.-à-d. celle du système). Plus précisément, cette approche cherche à réduire l'espace des comportements possibles en considérant un modèle explicite de l'environnement du système. En d'autres termes, il s'agit cette fois de « fermer » le système avec des ensembles choisis de cas d'utilisation de l'environnement auxquels il est sensé répondre, et uniquement ceux-ci. La réduction est basée sur une description formelle de ces cas d'utilisation, nommés *contextes*, avec lesquels le système interagit.

3.2 Identification des scénarios de contexte

Nous choisissons ici de spécifier l'environnement non plus sous la forme d'un seul processus global, mais comme un ensemble de scénarios explicites et séparés appelés contextes.

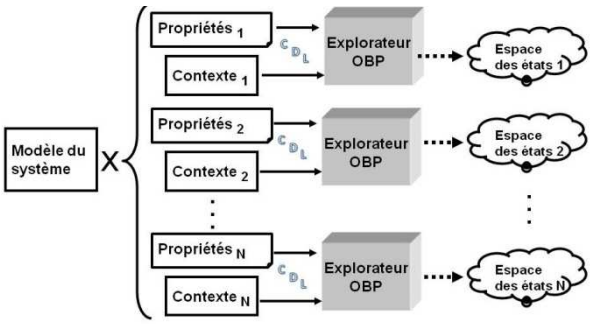

Figure 4 : Exploration avec identification de contextes séparés

Dans cette approche, l'environnement est décrit comme l'union de tous les contextes. Chaque contexte permet d'activer des sous-ensembles des comportements du modèle. Or les exigences à vérifier étant des propriétés de sûreté ou d'accessibilité, elles sont satisfaites si et seulement si elles le sont sur tous les contextes pris séparément [20]. Ce qui permet, lors de l'exploration, de ne plus explorer un « gros » espace d'états mais plusieurs (autant que de contextes) plus petits (figure 4). Cette approche « *diviser pour régner* » permet ainsi de mener les vérifications sur des systèmes de taille importante. Nous précisons ici ce principe mis en œuvre dans l'outil OBP.

Pour identifier les contextes et les spécifier formellement, l'utilisateur se base sur la connaissance qu'il a du système et sur son expertise permettant de spécifier de manière explicite son environnement. Il correspond généralement aux modes d'utilisation du système modélisé. Dans le contexte des systèmes embarqués réactifs, l'environnement de chaque composant d'un système est souvent bien connu. Il est donc plus efficace d'identifier cet environnement que de chercher à réduire l'espace des configurations du modèle du système à explorer.

L'objectif est de disposer de la description des sous-ensembles des comportements des acteurs de l'environnement ($Contexte_i$, $i \in [1..N]$) (figure 4) et des sous-ensembles de propriétés ($Propriétés_i$) associés à ces comportements. Cette identification des contextes peut déjà permettre de contourner l'explosion lors de l'exploration du modèle. 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 finis mais également complets.

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 considérons également que chaque contexte exprimé au départ est fini, c'est-à-dire que les scénarios décrits par ce contexte ne présentent pas de comportements itératifs infinis. Nous justifions ces hypothèses, en particulier dans le domaine de l'embarqué, par le fait que le concepteur d'un composant logiciel doit connaître précisément et complètement le périmètre (contraintes, conditions) de son utilisation pour pouvoir le développer correctement. Il serait nécessaire d'étudier formellement la validité de cette hypothèse de travail en fonction des applications ciblées. Nous n'abordons pas cet aspect qui donne lieu à un travail méthodologique spécifique à entreprendre. Néanmoins, dans l'approche actuelle, les comportements infinis de certaines entités de l'environnement doivent être inclus dans le modèle du système.

3.3 Partitionnement automatique des graphes de contexte

Lorsque la restriction des comportements du modèle, décrite précédemment, n'est pas suffisante, c'est-à-dire lorsque l'un des *Contexte*$_i$ conduit à un espace d'états qui explose, nous mettons en œuvre un second levier de réduction de l'espace des états. Chaque contexte qui conduit à une explosion est partitionné automatiquement et récursivement en un ensemble de sous-contextes plus réduits (figure 5).

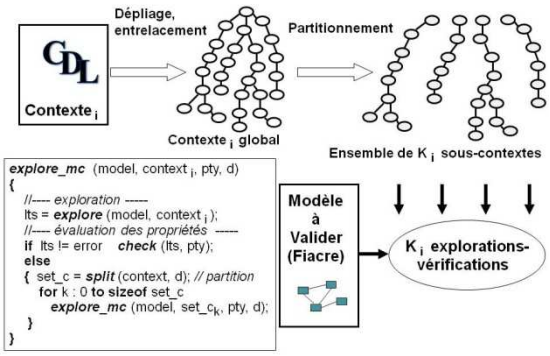

Figure 5 : Partitionnement d'un contexte et vérification pour chaque partition

Pour ce faire, nous mettons en œuvre un algorithme de partionnement récursif dans notre outil OBP. La figure 5 illustre la fonction *explore_mc()* pour l'exploration d'un modèle *model*, avec un contexte *context* et la vérification pour un ensemble de propriétés *pty*. Le contexte est représenté par un graphe acyclique. Ce graphe est composé avec le modèle lors de l'exploration. En cas d'explosion, ce contexte est automatiquement partitionné en plusieurs sous-graphes (avec la prise en compte d'un paramètre *d*, qui spécifie une profondeur du graphe à partir de laquelle la partition s'opère). Ce traitement récursif s'exécute jusqu'à ce que toutes les explorations aient été menées sans explosion.

De fait, pour chaque *Contexte*$_i$, nous transformons une vérification globale en K_i vérifications plus petites, où K_i est le nombre de sous-contextes obtenus après le partitionnement du *Contexte*$_i$. Au final, pour l'ensemble des contextes, cela revient à transformer N vérifications (autant que de contextes) en N' = $\sum_{i=1}^{N}$ Ki petites vérifications. Il faut noter que la technique de partitionnement mise en œuvre respecte le principe suivant : pour un contexte donné, l'union des exécutions, décrite par l'ensemble des sous-contextes générés par le partitionnement du contexte, inclut les exécutions décrites par ce contexte initial. Les propriétés de sûreté et de vivacité bornée sont donc préservées par le partitionnement du contexte comme démontré dans [20]. Nous montrons dans le paragraphe suivant le format de spécifications des contextes et des propriétés avec le langage CDL.

4. LE LANGAGE CDL

CDL [6] est un DSL[4] qui a deux objectifs : D'une part, la spécification des contextes, c'est-à-dire la description du comportement de l'environnement du modèle à valider et, d'autre part, la spécification des propriétés à vérifier. Un méta modèle de CDL a été défini, ainsi qu'une syntaxe et une sémantique formelles qui sont décrites [5] en termes de traces, s'inspirant des travaux de [12] et [22].

4.1 Description d'un contexte CDL pour notre cas d'étude

Pour la modélisation des contextes (c.-à-d., les Contextes$_i$ mentionnés au chapitre 3), un modèle CDL est structuré de manière hiérarchique. Doté d'une syntaxe textuelle, CDL est basé sur des diagrammes d'activités et de séquences. Il permet de décrire le comportement de plusieurs entités (nommées *acteurs*) composant l'environnement. Celles-ci s'exécutent en parallèle et interagissent avec le modèle du système par des communications asynchrones.

Un premier niveau déclare l'ensemble des acteurs constituant l'environnement du système et évoluant en parallèle les uns des autres. À ce niveau, un contexte CDL peut être représenté (de façon simplifiée) par une construction du type *A1||A2||...||An* où chaque A_i représente un acteur de l'environnement. Chaque A_i est ensuite détaillé sous la forme d'un diagramme d'activités, c'est-dire comme une composition de diagrammes de séquences ou de sous-diagrammes d'activités. Les

compositions possibles à ce niveau étant l'enchaînement (c.-à-d., la séquence) ou le choix (c.-à-d., l'alternative) entre deux ou plusieurs branches. Ensuite, chaque diagramme de séquences est décrit comme des interactions (*send* et *receive*) entre un acteur et le système. Formellement, un modèle CDL peut être considéré comme un ensemble de diagramme de séquences composés entre eux à l'aide de trois opérateurs : la séquence (« ; »), le parallèle (noté ||) et l'alternative (noté « [] »).

Lors de la compilation d'un modèle CDL, les diagrammes correspondant à chaque acteur sont dépliés (mise à plat de chaque boucle finie) puis entrelacés entre eux conformément à la sémantique du parallélisme de Fiacre. L'entrelacement de l'ensemble des diagrammes de séquences, décrivant le comportement du contexte, génère un graphe représentant toutes les exécutions des acteurs de l'environnement considéré. Ce graphe est ensuite partitionné, de manière à générer un ensemble de sous-graphes correspondant aux sous-contextes, comme mentionné en 3.3. Lors de l'exploration par le vérificateur, chaque sous-graphe est composé (figure 5) avec le modèle à valider et les propriétés sont vérifiées sur le résultat de cette composition.

Pour le cas d'étude traité ici, le contexte CDL décrit les interactions entre le *DCM* et le processus *Controller*. Le *DCM* est l'unique acteur CDL dans l'environnement du pacemaker. Les interactions sont décrites sous la forme d'alternatives et de séquences qui modélisent les scénarios envisagés, c'est-à-dire les combinaisons de valeurs fournies au *Controller*. La figure 6 représente graphiquement le modèle CDL du comportement du *DCM*. Il décrit l'envoi au processus *Controller* de la valeur du mode puis une valeur pour chacun des paramètres *ARP*, *VRP*, *LRL* et *AVDelay*. Selon ce contexte, le *DCM* peut exécuter l'un des sept diagrammes de séquences *SetModeX* (où *X* est *ARP*, *VRP*, *AVDelay* ou *LRL* et l'opération « alternative » est représentée graphiquement par un losange).

Chaque diagramme *SetModeX* modélise l'envoi d'un ordre de changement de mode du *DCM* vers le pacemaker. Par exemple, *SetModeAOO* est un diagramme de séquences contenant l'unique interaction « *chgParam (mode, AOO)* » de *DCM* vers *Controller* (figure 6). Après exécution de ce changement de mode, *DCM* continue par l'exécution de zéro, un, deux ou trois diagrammes de séquences *SetValX* selon la branche prise lors de la première alternative. Chaque *SetValX* est un diagramme de séquences qui modélise toutes les alternatives possibles pour l'affectation d'une valeur au paramètre *X*. Par exemple, le diagramme *setValARP* modélise une alternative entre 36 affectations possibles du paramètre *ARP*. Le modèle CDL de la figure 6 représente donc l'ensemble des 62 498 combinaisons possibles en entrée du pacemaker.

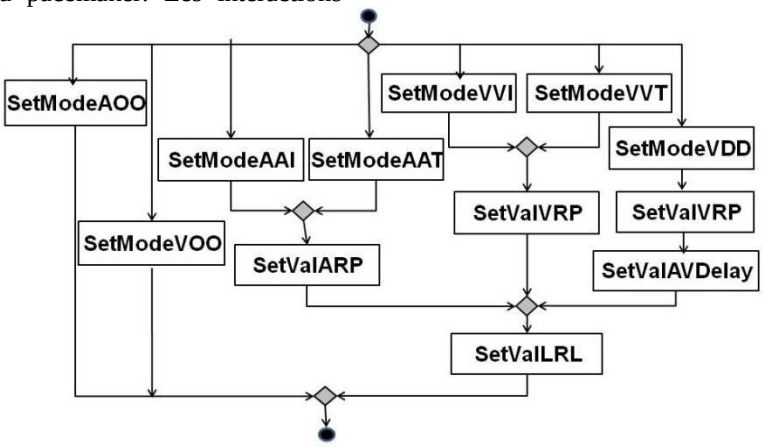

Figure 6 : Modèle CDL du *DCM*

La version textuelle de l'ensemble des scénarios illustré par la figure 6 est illustrée listing 1. L'opérateur "[]" spécifie l'alternative et le ";" la séquence. Le scénario *allModes* référence la propriété *R1* correspondant à l'observateur à vérifier.

```
cdl  allModes is {
        properties R1;
```

```
{
  {  set_mode_AOO [ ] set_mode_VOO
     [ ]
     { {
        {  { set_mode_AAI [] set_mode_AAT };
           set_val_ARP  }
        [ ]
        {  { set_mode_VVI [] set_mode_VVT };
           set_val_VRP }
        [ ]
        { set_mode_VDD; set_val_VRP;
```

```
        set_val_AVDelay
      }
  }; set_val_LRL } };
} };
```

Listing 1 : Le modèle CDL (version textuelle) pour tous les modes

Le scénario *allModes* référence des activités telles que :

- set_val_ARP équivalent à : setARP_150 [] . . . [] setARP_500.
- set_val_VRP équivalent à : setVRP_150 [] . . . [] setVRP_500.
- set_val_AVDelay equivalent à: setAVDelay_70 [] . . . [] setAVDelay_300.
- set_val_LRL équivalent à : setLRL_30 [] . . . [] setLRL_175.
-

4.2 Formalisation des propriétés CDL

CDL permet également de spécifier des propriétés sous la forme d'invariant ou d'observateur. La spécification des propriétés CDL sont établies à partir d'une formalisation de prédicats et d'événements. Un prédicat référence des variables et s'exprime comme suit :
*predicate pred1 is { {proc}*1 : *v = value}* signifie que *pred1* est vrai si la variable *v* de la première instance du processus *proc* est égal à la valeur *value*. Un prédicat peut aussi référencer un état d'un processus tel que par exemple : *predicate pred2 is {{proc}*1*@stateX}* signifie que *pred2* est vrai si la première instance du processus *proc* est dans l'état *stateX*. Un prédicat peut être une expression booléenne combinant plusieurs prédicats.
Par exemple, dans notre cas d'étude, le prédicat *p_startingCycle* est défini à partir des états *TP_StP*, *AsyncPac*, *InPac_SP*, *TP_WP* de l'automate de *Cardio*. Ce qui donne :

predicate p_startingCycle is {

({cardio}1@TP_StP or {cardio}1@AsynPac

or {cardio}1@InPac_SP or {cardio}1@TP_WP) }

Un événement CDL peut être défini à partir d'un prédicat. Par exemple, l'événement *startingCycle* est spécifié come suit :

event startingCycle is { p_startingCycle becomes true }

Un événement CDL peut aussi référencer une opération de communication. Par exemple, l'événement *stopBrady* est spécifié par :

event stopBrady is { receive msg_stopBrady from {Controller}1 to {cardio}1 }

qui spécifie une réception d'un message *msg_stopBrady* par le processus *cardio* et provenant du processus *Controller*.

Enfin, l'observateur de l'exigence *R1* s'écrit de la manière suivante :

property R1 is { clock c;

start -- / / startingCycle / c:=0 -> wait ;

wait -- c <= LRL / / stopBrady / c:=-1 -> start ;

wait -- c <= LRL / / startingCycle / c:=0 -> wait ;

wait -- c > LRL / / / c:=-1 -> Reject }

4.3 Exploitation des modèles CDL dans l'explorateur OBP

L'outil OBP prend en entrée les modèles CDL pour générer des données adéquates pour l'explorateur OBP. Une analyse d'accessibilité est menée sur le résultat de la composition entre un graphe de contexte généré, un ensemble d'observateurs et le modèle. OBP récupère les résultats de l'exploration et les formate pour les rendre compréhensibles par l'utilisateur. La partition du contexte par OBP en un ensemble de graphes permet d'aboutir, lors de la composition, à des systèmes de transitions de taille limitée rendant possible l'analyse d'accessibilité.

4.4 Résultats

Nous présentons les résultats de la vérification sur le pacemaker en mettant en œuvre les contextes CDL et la technique du partionnement des graphes de contextes. Le tableau 2 présente les résultats pour la vérification de l'observateur correspondant à l'exigence *R1*, pour tous les modes, avec l'exploitation du modèle CDL du *DCM*. On constate que la vérification termine en environ 6 heures y compris pour le modèle complet du *DCM* offrant 62 498 combinaisons possibles (dernière ligne du tableau). L'espace d'états exploré contient environ 535 millions de configurations et nécessite un découpage automatique en 940 sous-contextes. L'évolution du temps de vérification est montrée par une succession d'expérimentations réalisée sur des contextes de plus en plus grands (de 352 à 62 498 combinaisons). Elle est exponentielle, mais reste faisable en pratique.

Nombre de combinaisons	Nombre de Sous-contextes générés	Nombre de Configurations explorées	Nombre de Transitions explorées	Temps d'exploration (secondes)

352	7	2 680 017	3 362 387	92
1 282	14	10 179 272	13 069 412	437
15 202	344	126 267 775	16 5504 813	6 355
39 202	648	329 522 688	434 009 683	14 226
62 498	940	535 871 149	706 896 539	22 238

Tableau 2 : Vérification de l'exigence R1 avec OBP et l'exploitation des contextes

5. BILAN ET CONCLUSION

5.1 Bilan

L'identification des contextes CDL et la technique de partitionnement permettent de réduire considérablement l'espace d'exploration des modèles sans lesquels l'exploration de tous les comportements est impossible. Plus précisément, nous avons montré que le partitionnement de l'environnement, opéré par OBP, en plusieurs cas d'utilisation séparés mais couvrant complètement l'ensemble des comportements de l'environnement (c.-à-d., formant une partition au sens mathématique) permet de repousser les limites de l'explosion combinatoire. Cette démarche de partitionnement de l'environnement peut être déclenchée automatiquement chaque fois que la barrière de l'explosion combinatoire est rencontrée, permettant ainsi la vérification de propriétés sur de grands modèles. Nous avons montré que l'application de l'ensemble des 62 498 combinaisons des valeurs des paramètres associées aux modes choisis peut mener à une vérification de l'exigence considérée.

De plus, l'apport des modèles CDL est d'offrir un cadre formel pour décrire les exigences exprimables sous forme d'invariants ou d'automates observateurs. Dans des applications industrielles, la description des contextes interagissant avec le modèle à valider est souvent informelle, parfois incomplète. L'approche 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. Cette formalisation est basée sur des diagrammes d'activités et des diagrammes de séquences qui sont d'un accès aisé pour un ingénieur. Conceptuellement, les principes de CDL peuvent donc être implantés dans d'autres formalismes plus standard comme les diagrammes d'activités et de séquences UML.

Dans notre illustration, nous avons considéré le *DCM* comme le seul acteur de l'environnement. Du fait de n'avoir, dans ce cas, qu'un acteur unique, l'intérêt du partitionnement automatique peut sembler limité. Ce n'est pas le cas lorsque l'environnement est composé de plusieurs acteurs. En effet, OBP construit à partir du comportement de chaque acteur, un graphe de l'ensemble des comportements des acteurs en considérant l'entrelacement des comportements. Ce graphe est ensuite partitionné avec la technique décrite dans cet article.

Nous pourrions mener des vérifications sur des modèles plus complexes, par exemple pour un nombre de combinaisons beaucoup plus important. Le temps d'exploration et de vérification augmenterait en conséquence. Pour accélérer les vérifications, il faudrait accroître les performances de l'outillage en augmentant d'une part la mémoire pour limiter les opération de partitionnement des contextes et, d'autre part, en parallélisant le traitement des contextes sur un réseau de machines [5].

5.2 Conclusion

Cet article décrit une technique de réduction de l'espace d'exploration de modèles pour la vérification formelle de propriétés. Le principe est d'amener l'explorateur de modèles à concentrer ses efforts non plus sur l'exploration de l'espace complet des comportements, mais sur une restriction pertinente de ce dernier pour la vérification de propriétés spécifiques. La vérification de propriétés devient alors possible sur l'espace d'états ainsi réduit. Nous avons illustré cette approche sur un modèle simplifié de contrôleur de pacemaker. L'approche décrite cherche à réduire l'espace des comportements possibles en considérant un modèle explicite de l'environnement du système formalisé sous la forme d'une union de contextes. Ce modèle est décrit à l'aide du langage CDL qui permet de décrire les interactions de l'environnement avec le modèle à valider. Un contexte ainsi formalisé peut être exploité par OBP, explorateur de modèles Fiacre, et partitionné en des sous-contextes qui sont composés avec le modèle à valider. Ce partitionnement permet de réduire le nombre de comportements du modèle à explorer à chaque vérification, et ainsi de faire reculer l'explosion combinatoire. Lors des explorations, nous mettons en œuvre et vérifions des observateurs. Nous avons illustré cette technique pour un ensemble d'interactions entre le *DCM* et le contrôleur du pacemaker.

6. BIBLIOGRAPHIE

[1] R. Alur, R. K.. Brayton, T. A. Henzinger, S. Qadeer et S. K. Rajamani : Partial-order reduction in symbolic state space exploration ; Computer-Aided Verification, vol. 1254, Springer Verlag, LNCS, pp. 340-351, 1997.

[2] B. Berthomieu, P.-O. Ribet et F. Verdanat : The tool TINA - Construction of abstract state spaces for Petri nets and time Petri nets ; International Journal of Production Research, vol. 42, pp. 2741-2756, juillet 2004.

[3] D. Bosnacki et G. J. Holzmann : Improving Spin's partial-order reduction for breadth-first search ; SPIN, vol. 3639, pp. 91-105, 2005.

[4] E. Clarke, E. Emerson et A. Sistla : Automatic verification of finite-state concurrent systems using temporal logic specifications ; ACM Trans. Program. Lang. Syst., vol. 8, n° 2, pp. 244-263, 1986.

[5] P. Dhaussy, F. Boniol et J.-C. Roger : Reducing state explosion with context modeling for model-checking ; 13th IEEE International High Assurance Systems Engineering Symposium (Hase'11), Boca Raton, États-Unis, 2011.

[6] P. Dhaussy et J.-C. Roger : CDL (Context Description Language) : Syntaxe et sémantique ; rapport technique, disponible sur http://www.obpcdl.org, ENSTA-Bretagne, 2011.

[7] E. A. Emerson, S. Jha et D. Peled : Combining partial order and symmetry reductions ; in E. Brinksma (ed.), Tools and Algorithms for the Construction and Analysis of Systems, vol. 1217, Springer Verlag, LNCS, Enschede, Pays-Bas, pp. 19-34, 1997.

[8] P. Farail., P. Gaufillet, F. Peres, J.-P. Bodeveix, M. Filali, B. Berthomieu, S. Rodrigo, F. Vernadat, H. Garavel et F. Lang : FIACRE: an intermediate language for model verification in the TOPCASED environment, European Congress on Embedded Real-Time Software (ERTS), SEE, Toulouse, janvier 2008.

[9] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu et M. Sighireanu : CADP: A protocol validation and verification toolbox ; CAV '96 : Proceedings of the 8th International Conference on Computer-Aided Verification, Springer-Verlag, London, GB, pp. 437-440, 1996.

[10] P. Godefroid, D. Peled et M. G. Staskauskas : Using partial-order methods in the formal validation of industrial concurrent programs ; International Symposium on Software Testing and Analysis, S. J. Zeil et W. Tracz, ACM Press, New York, San Diego, CA, pp. 261-269, janvier, 1996.

[11] N. Halbwachs, F. Lagnier et P. Raymond : Synchronous observers and the verification of reactive systems ; in M. Nivat, C. Rattray, T. Rus, G. Scollo (eds), Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Workshops in Computing, Springer Verlag, Twente, pp. 83-96, juin, 1993.

[12] O. Haugen, K. E. Husa, R. K. Runde et K. Stolen : STAIRS towards formal design with sequence diagrams ; Software and System Modeling, vol. 4, n° 4, pp. 355-357, 2005.

[13] T. Henzinger, Z. Manna et A. Pnueli : Timed transition systems ; in J.W. de Bakker, C. Huizing, W.-P. de Roever et G. Rozenberg (eds), Real-Time: Theory in Practice, Proceedings of REX Workshop 1991 (REX'91), vol. 600 of Lecture Notes in Computer Science, Springer-Verlag, pp. 226-251, juin, 1992.

[14] G. J. Holzmann : The model checker SPIN ; IEEE Transactions on Software Engineering, vol. 23, n° 5, pp. 279-295, mai 1997.

[15] K. G. Larsen, P. Pettersson et W. Yi : UPPAAL in a nutshell ; International Journal on Software Tools for Technology Transfer, vol. 1, n° 1-2, pp. 134-152, 1997.

[16] K. L. Mc Millan et D. K. Probst : A technique of state space search based on unfolding ; Formal Methods in System Design, vol. 6, pp. 45-65, janvier, 1995.

[17] S. Park et G. Kwon : Avoidance of state explosion using dependency analysis in model checking control flow model : Proceedings of the 5th International Conference on Computational Science and its Applications (ICCSA '06), vol. 3984, Springer-Verlag, LNCS, pp. 905-911, 2006.

[18] D. Peled : Ten years of partial order reduction ; CAV '98: Proceedings of the 10th International Conference on Computer-Aided Verification, Springer-Verlag, pp. 17-28, 1998.

[19] J.-P. Queille et J. Sifakis : Specification and verification of concurrent systems in CESAR ; Proceedings of the 5th Colloquium on International Symposium on Programming, Springer-Verlag, Londres, GB, pp. 337-351, 1982.

[20] J.-C .Roger : Exploitation de contextes et d'observateurs pour la validation formelle de

modèles ; Thèse de Doctorat, ENSIETA, Université de Rennes I, décembre, 2006.

[21] A. Valmari : Stubborn sets for reduced state space generation ; Proceedings of the 10th International Conference on Applications and Theory of Petri Nets, Springer-Verlag, Londres, GB, pp. 491-515, 1991.

[22 J. Whittle : Specifying precise use cases with use case charts ; in MoDELS'06, Satellite Events, pp. 290-301, 2006.

NOTES

1- OBP est libre d'accès sur le site : http://www.obpcdl.org.

2- Le modèle complet Fiacre du pacemaker ainsi que les descriptions CDL du DCM peuvent être trouvés sur le site http://www.obpcdl.org.

3- Les tests sont effectués sur une configuration PC Linux 32 bits, 3 GO de RAM, avec OBP v. 1.3.4.

4- Domain Specific Language

Luka Leroux est ingénieur de recherche. Apres un Master recherche à l'université de Rennes I, il a rejoint en 2011 l'équipe du pôle STIC de l'ENSTA-Bretagne. Ses recherches sont orientées dans le domaine de modélisation des logiciels embarqués et les techniques de vérifications formelles d'exigences.

Philippe Dhaussy est directeur du pôle STIC de l'ENSTA-Bretagne. Son expertise et ses recherches sont orientées vers l'Ingénierie Dirigées par les Modèles et les techniques de validation formelle pour le développement de logiciels embarqués et temps réel. Diplômé ingénieur (1978) de l'Institut Supérieur d'Électronique du Nord (ISEN), il a obtenu le titre de docteur (1994) à Telecom Bretagne. Après un parcours industriel (1980-1991), il a rejoint l'ENSTA-Bretagne et a contribué à la création du groupe de recherche en modélisation de systèmes et logiciels embarqués, intégré au laboratoire Lab-STICC UMR CNRS 6285.

Frédéric Boniol a obtenu un diplôme d'ingénieur de Supaero en 1987, puis un doctorat en informatique fondamentale en 1997. Après avoir été professeur à l'IRIT jusqu'en 2008, il a dirigé une équipe de recherche sur les architectures embarquées à l'ONERA. Il conduit depuis des travaux recherche sur la modélisation et la vérification de systèmes critiques embarqués, en liaison avec les industriels du domaine (Airbus, Thales...) et les principales équipes académiques (IRIT, LAAS, ENSTA-Bretagne, CEA...).