Vérification formelle de propriétés :

Application de l'outil OBP au cas d'étude CCS

Philippe Dhaussy, Luka Le Roux, Ciprian Teodorov

Résumé :

Malgré le niveau élevé d'automatisation des techniques de *model-checking*, celles-ci se trouvent limitées par le problème de l'explosion du nombre d'états générés lors de l'exploration de modèles de taille importante. Pour répondre à ce défi, nous proposons une technique de vérification prenant en compte la spécification explicite de l'environnement (contexte) dans lequel le modèle du système est plongé lors de son exploration par le vérifieur. 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.

Dans cet article, nous appliquons cette technique à la vérification du régulateur de vitesse (CCS ou *Control Cruise Control*) décrit dans l'article précédent [21]. L'asynchronisme intrinsèque de ce système rend les approches traditionnelles presque impossibles. En exploitant les contextes, nous montrons que la vérification devient possible en s'appuyant sur deux stratégies efficaces d'optimisation basées sur les propriétés structurelles des contextes. Une première qui met en œuvre un fractionnement automatique du contexte par une décomposition récursive de son espace d'état. Une seconde qui exploite la mémoire disque externe optimisant l'utilisation de la mémoire lors de l'exploration exhaustive du modèle. Dans le cas du système CCS, cette approche a permis l'analyse d'espace d'états 5 fois plus grands que les approches traditionnelles.

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

1. INTRODUCTION

Les techniques de vérification formelle de propriétés ont été largement explorées et, en particulier, celles dans le domaine du *model-checking* [4,20]. Celles-ci ont été fortement popularisées grâce à leur faculté d'exécuter automatiquement des preuves de propriétés et au développement de nombreux outils (*model-checkers*) [2,10,13,14]. Le principe général de cette technique 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 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 CCS qui est connecté aux capteurs de l'environnement (boutons de l'IHM, pédales de frein et d'accélérateur, vitesse) et actionneur (commande de vitesse vers le moteur).

Deuxièmement, une complexité interne est liée à l'architecture logicielle du système embarqué lorsque celle-ci est composée de plusieurs processus logiciels interagissant en parallèle. Dans les deux cas (interne et externe), le parallélisme d'entités externes ou de processus interne au système conduit à une explosion combinatoire du nombre de comportements possibles du modèle à analyser.

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 [1,3,8,11,13,17,18,19]. 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 décrivons dans cet article une technique complémentaire de vérification prenant en compte la spécification explicite de l'environnement (contexte) dans lequel le modèle du système est plongé lors de son exploration par le vérifieur. La technique s'appuie sur l'exploration partielle du modèle, qui permet dans certains cas pertinents, de limiter l'explosion combinatoire. Cette technique de vérification orientée contexte a été récemment

introduit dans [5, 6, 16] comme une technique de décomposition état-espace qui permet la vérification compositionnelle des exigences. Cette technique permet de réduire l'ensemble des comportements possibles (et donc l'espace des états) en fermant le modèle à explorer avec un environnement fini acyclique. La spécification explicite et formelle de cet environnement permet au moins trois axes de décomposition différents :

- l'environnement peut être décomposé en des contextes, chacun isolant ainsi différents modes de fonctionnement du modèle;
- les contextes peuvent être partitionnés automatiquement décomposant la vérification globale en un ensemble de vérifications indépendantes;
- les propriétés à vérifier sont spécifiques à certains contextes environnementaux.

Nous développons les principes de cette approche dans le cadre d'une modélisation en langage Fiacre [9] des modèles à valider et de l'exploitation d'un langage, nommé CDL (*Context Description Language*) [7], pour la description des cas d'utilisation de l'environnement considérés pour les validations. Cette approche est instrumentée à l'aide d'un outil, nommé OBP[1] (*Observer-Based Prover*) [5] 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 (Figure 1).

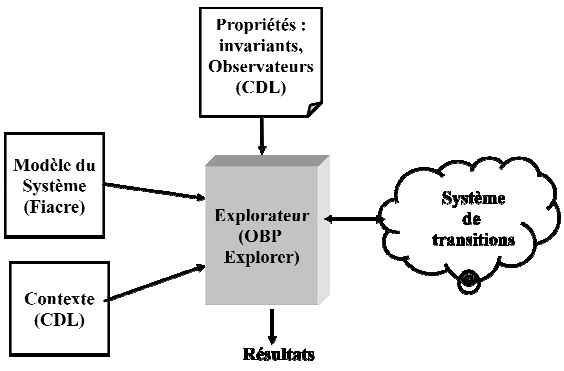

Figure 1 : L'outil OBP.

Nous illustrons, dans cet article, l'apport de cette approche sur le cas d'étude du régulateur de vitesse automobile CCS (*Control Cruise Control*) décrit précédemment [15]. En utilisant cette approche, nous avons vérifié trois exigences importantes du CCS et identifié une erreur potentielle dans le modèle qui pourrait conduire à des situations très dangereuses.

[1] OBP est librement disponible sur le site http://www.obpcdl.org.

En outre, nous mettons en évidence l'apport de l'approche en soulignant la possibilité d'analyse d'espace des états 4,78 plus grand que dans les approches traditionnelles. Ce résultat est rendu possible par la complémentarité des deux stratégies efficaces d'optimisation basées sur les propriétés structurelles des contextes. Une première qui met en œuvre un fractionnement automatique du contexte par une décomposition récursive de son espace d'état. Une seconde qui exploite la mémoire disque externe optimisant l'utilisation de la mémoire lors de l'exploration exhaustive du modèle.

L'article est organisé comme suit. Dans la section 2, nous présentons l'approche de vérification s'appuyant sur les contextes ainsi que les deux techniques d'analyse adressant le problème de l'explosion de l'espace des états. En section 3, nous appliquons le langage CDL au codage des propriétés du système CCS ainsi que celui de l'environnement. Les résultats des vérifications sont présentés Section 4. Enfin, nous concluons cet article Section 5.

2. LA VÉRIFICATION BASÉE SUR LES CONTEXTES

2.1 Principes de l'approche

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. Lors de la vérification des propriétés, tous les comportements possibles du système sont explorés exhaustivement et les propriétés sont évaluées sur ce *SdT* vraies ou fausses. La vérification 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). En effet, en raison de la croissance exponentielle des états du modèle par rapport au nombre de composants en interaction, très souvent le nombre de configurations accessibles est trop grand pour qu'elles puissent être contenues en mémoire.

Pour remédier à ce problème, les concepteurs de systèmes doivent configurer, explicitement et manuellement, le modèle de façon à limiter ses comportements à ceux qui sont pertinents au regard des propriétés spécifiées à vérifier. Ce processus est

fastidieux et peut générer des erreurs. Il pose aussi un certain nombre de défis méthodologiques concernant la gestion des différentes versions du modèle devant être conservées et maintenues.

L'approche (voir les détails dans [5, 6, 16]) que nous proposons, basée sur les contextes, se concentre sur la modélisation explicite de l'environnement comme une composition de plusieurs contextes, qui sont chacun composé avec le modèle du système. Les exigences sont associées et vérifiées pour chaque contexte qui correspond aux conditions environnementales dans lesquelles ces exigences doivent être satisfaites. De plus, une technique de réduction automatique de l'espace des états guidée par les contextes *(Automatic Context Splitting)* est exploitée pour faire reculer plus loin les limites de l'analyse d'accessibilité. Tous ces développements sont mis en œuvre dans l'outil OBP.

OBP met également en œuvre un nouvel algorithme [21] d'analyse exhaustive qui optimise la consommation mémoire (*Context-directed semi-external reachability analysis*). Le principe est le suivant : Lors de la construction du graphe des états, seules les configurations utiles sont gardées en mémoire, les autres étant stockées sur disque, comme mentionné dans [21]. L'implantation de cet algorithme, nommé *PastFree[ze]*, est rendue possible grâce à la structure acyclique du contexte et à son possible partitionnement. Les nœuds du graphe du contexte devenus inutiles durant l'exploration peuvent être libérés de la mémoire et sauvegardés sur le disque.

En outre, cette approche a un impact méthodologique car elle permet de découpler le modèle de son environnement, lui permettant d'être traité séparément du modèle. La réduction des comportements du système prenant en compte le contexte est particulièrement intéressante dans le cas de systèmes embarqués complexes, tels que l'automobile et l'avionique, car ceux-ci présentent des modes de fonctionnement bien identifiés avec des propriétés spécifiques associés à ces modes. Dans notre approche, nous proposons de capturer ces contextes dans le langage CDL.

2.2 Modélisation CDL des contextes

CDL[2] [7] est un DSL[3] 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.

Pour la modélisation des contextes, 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 parallélisme (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. 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 automatiquement partitionné comme décrit en [5,6,16].

2.3 Modélisation CDL des propriétés

CDL permet de spécifier des propriétés sous la forme d'invariant ou d'observateur. La spécification des propriétés CDL est établie 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.

[2] La syntaxe et la sémantique de CDL sont décrites dans http://www.obpcdl.org.
[3] Domain Specific Language.

Un événement CDL peut être défini à partir d'un prédicat. Par exemple, l'événement *e* est spécifié à partir du prédicat *pred* comme suit :

***event** e **is** { pred **becomes** true }*

Un événement CDL peut aussi référencer une opération de communication entre 2 processus. Par exemple, l'événement *e_recv* est spécifié par :
event** e_recv **is

*{ **receive** msg **from** {ProcA}1 **to** {ProcB}1 }*

e_recv spécifie une réception d'un message *msg* par la première instance du processus *ProcB* et provenant de la première instance du processus *ProcA*.
Enfin, les propriétés peuvent être exprimées sous la forme d'un automate observateur. Un automate observateur [12] 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 faussée. 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*. Nous illustrons Figure 3 un observateur et son codage en CDL en Listing 2.

3. MODÉLISATION CDL POUR LE CCS

L'architecture identifiée (Figure 2) pour la vérification est un peu différente de celle présentée dans l'article [15] pour la modélisation du CCS. Mais cette différence n'a pas d'impact dans le processus de vérification mené.

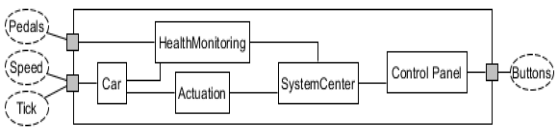

Figure 2 : Architecture du CCS.

Un nouveau processus est introduit (*Car*) connecté au CCS via les ports d'Actuation et de Health Monitoring. Ce nouveau processus est responsable de la diffusion de la vitesse courante sur réception d'un evènement *evtTick* venant de l'environnement.

3.1. Modélisation des exigences

Les exigences qui ont été décrites dans [15] et qui doivent être vérifiées sur le modèle du CCS sont :

- ***Req1***: ***Après** la détection d'un événement induisant un désengagement et **tant que** le système n'est pas réengagé, le CCS **ne doit pas** tenter d'ajuster la vitesse du véhicule.*
- ***Req2***: *La vitesse de croisière **ne doit pas** être inférieure à 40km/h ou supérieure à 180km/h.*
- ***Req3***: ***Tant que** le système est engagé, la vitesse cible **doit être** considérée comme définie.*

Req1 peut être codé à l'aide de l'automate d'observateur présentée dans la figure 3.

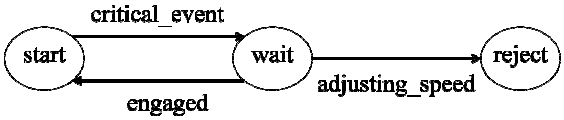

Figure 3 : Observateur pour la propriété Req1.

Pour encoder cet observateur en CDL, nous spécifions des événements qui sont nécessaires au déclenchement des transitions.

```
predicate disengageIsRequested is {
      HealthMonitoring@DisengageRequested }

event critical_event is {
            disengageIsRequested becomes true }

event adjusting_speed is {
                  send any from Actuation to Car }

event engaged is {
          SystemCenter@Engaged becomes true }
```

Listing 1 : Déclaration CDL des évènements

Dans Listing 1, *disengageIsRequested* est un prédicat évalué à *vrai* si le processus *HealthMonitoring* est dans l'état *DisengageRequested*. L'évènement *critical_event*, observable dans l'explorateur OBP, spécifie un front montant du prédicat *disengageIsRequested*. Il exprime que le processus de *HealthMonitoring* vient d'entrer dans l'état *DisengageRequested*. *adjusting_speed* exprime un événement observable qui est détecté lorsque le processus *Actuation* envoie un message vers le processus *Car*. Dans notre modèle, les seuls messages qui passent par ce canal sont les consignes de vitesse. En fait, cette

demande correspond à une tentative de *Actuation* pour contrôler la vitesse. En utilisant ces événements, l'automate observateur Figure 3 est spécifié Listing 2.

```
property Req1 is {
   start  -- critical_event --> wait;
   wait   -- adjusting_speed --> reject;
   wait   -- engaged --> start
}
```

Listing 2 : Automate CDL de la propriété Req1.

Req2 peut être encodée sous forme d'invariant (Listing 3) en déclarant le prédicat *targetSpeedIsUnSet*.

```
predicate targetSpeedIsUnSet is {
   Actuation@UnSet    or
   Actuation@UnsetSetting
}
predicate Req2 is {
   (Actuation:targetSpeed >= 40 and
    Actuation:targetSpeed <= 180) or
    targetSpeedIsUnSet
}
```

Listing 3 : Spécification de Req2.

L'encodage de *Req3* sous la forme d'invariant est présenté Listing 4. Il s'agit d'un prédicat faussé si la vitesse de croisière n'est pas définie alors que le système est engagé.

```
predicate Req3 is {
      not (targetSpeedIsUnSet and
           SystemCenter@Engaged)
}
```

Listing 4 : Spécification de Req3.

3.2. Modélisation de l'environnement

La modélisation de l'environnement consiste en la spécification des entités qui interagissent avec le système CCS. Pour modéliser cet environnement, nous identifions les comportements des entités sous la forme d'acteurs CDL. L'environnement est construit à partir de deux acteurs modélisant un scénario nominal et un générateur de *ticks* de synchronisation comme expliqué ci-dessous. Un troisième acteur, le perturbateur, est identifié et a pour rôle d'interagir avec le CCS et d'en perturber le fonctionnement. Par exemple, il agit en simulant des changements de vitesse du véhicule, dans la plage des valeurs autorisées ou non, des pressions sur les pédales, accélération ou débrayage et des pressions sur les boutons de l'IHM. Le perturbateur simule ainsi des comportements inattendus mais possibles de l'environnement.

Lors de la modélisation de l'environnement, la méthodologie impose d'abord que le contexte spécifie tous les comportements à prendre en considération pour une propriété donnée. Mais, il doit être assez restreint pour que l'exploration exhaustive soit possible lors de sa composition avec le modèle.
Le scénario nominal (Figure 4) peut être considéré comme un cas d'utilisation du CCS qui couvre les fonctionnalités pour lesquelles les propriétés que nous cherchons à vérifier sont impliquées.

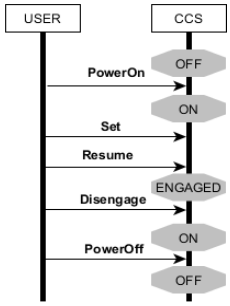

Figure 4 : Scénario nominal.

Le comportement du perturbateur est encodé (Listing 5) par une activité CDL (*activity*) comme suit :

```
activity perturbator is {
      evtSetSpeed_40  []  evtSetSpeed_180
 []    evtSetSpeed_0 [] evtSetSpeed_200
 []    evtBrakePressed [] accelerate
 []  { evtBtnSet; evtBtnResume; evtBtnDisengage }
}
```

Listing 5 : Description CDL du perturbateur.

Le codage de ce perturbateur référence les différentes capacités de l'environnement, dont l'envoi de consignes de vitesse (par exemple *evtSetSpeed_40*), des pressions sur les pédales (par exemple *evtBrakePressed)* ou boutons de l'IHM (*evtBtnResume*).

Un *tick* est un événement envoyé au processus *Car* pour déclencher la diffusion de la vitesse courante vers les composants impliqués dans le CCS. En d'autres termes, chaque *tick* permet au système de «lire» une fois la vitesse courante. Le scénario nominal utilisé référence deux changements de vitesse. Le perturbateur peut émettre une troisième demande de changement de vitesse, ce qui permet de tester que le système soit en mesure de réagir correctement. La génération de 3 *ticks* est donc nécessaire pour assurer que ces différents changements de vitesse soient diffusés et pris en compte par le système. L'émission d'un quatrième *tick* permet d'observer le comportement du système dans le cas où la vitesse courante reste constante. Nous nous servons d'une variable pour paramétrer le nombre de *ticks* pris en considération. En effet, nous chercherons, en modifiant sa valeur, à complexifier artificiellement les comportements du

modèle pour montrer l'efficacité des différentes politiques d'exploration basées sur les contextes.
Une fois le scénario nominal, le perturbateur et la génération des *ticks* composés, nous codons un contexe m*yContext* (Listing 6) qui simule l'ensemble des exécutions entrelacées, illustré Figure 5, entre les actions élémentaires du scénario nominal, celles du perturbateur et la génération des *ticks*.

```
cdl myContext is {
        properties req1
        assert req2, req3
        init is  {   evtBtnStart }
        main is {    basic_scenario
                  || perturbator
                  || loop 4 evtTick }
}
```

Listing 6 : Description CDL du contexte.

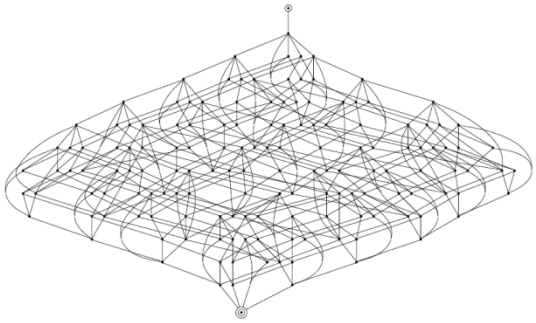

Figure 5 : Graphe des exécutions du contexte.

4. VÉRIFICATION

Nous présentons les résultats obtenus pour la vérification des trois exigences présentées précédemment, mettant l'accent sur l'apport de l'approche orientée par les contextes. Par rapport aux algorithmes de recherche classique de recherche en largeur (BFS), l'utilisation seule de l'algorithme PastFree[ze] a permis l'analyse d'un espace des états 2,4 fois plus grand. L'utilisation conjointe de PastFree[ze] [21] et de la technique de partitionnement automatique a permis l'analyse d'un espace des états 4,78 fois plus grand espace d'état. Les résultats présentés dans cette étude ont été obtenus sur un ordinateur 64 bits Linux, avec un processeur Intel Xeon 3.60 GHz, 64 Go de mémoire RAM et la version d'OBP 1.4.6[4].

Lors des premières vérifications, au cours de l'exploration, l'observateur *Req1* atteint l'état *reject*. Cela signifie que la propriété n'était pas vérifiée. Cela se produit en raison d'un défaut dans le modèle qui s'explique comme suit : Lors de la réception d'un événement *tick*, la voiture diffuse sa vitesse courante à la fois au processus Actuation et à Health Monitoring. Les deux processus réagissent alors, le premier en renvoyant une demande d'ajustement d ela vitesse, le second en détectant cet événement comme critique et en demandant un désengagement. Le modèle a donc été corrigé de manière à satisfaire *Req1*. Une possibilité était de modifier le système de façon à ce qu'Actuation n'ajuste la vitesse courante que si elle est comprise dans les bornes de valeur autorisées.

Exploration avec l'algorithme BFS

La figure 6 présente les résultats de l'exploration en termes de nombres de configurations, de transitions et de temps d'exploration (*time)* obtenu en faisant varier de 4 à 11 le nombre de *ticks* généré par le contexte présenté Listing 6. L'algorithme d'analyse d'accessibilité traditionnelle BFS ne permet pas une exploration exhaustive au-delà de 9 *ticks*.

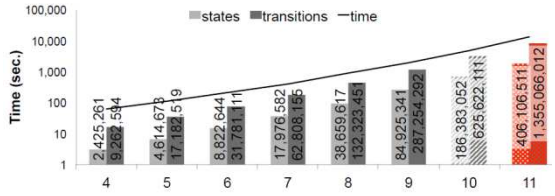

Figure 6 : Résultat des explorations pour différents contextes (de 4 à 11 *ticks*).

Exploration avec l'algorithme PastFree[ze]

L'analyse PastFree[ze] permet d'explorer avec succès, pour 10 *ticks*, un espace des états 2,2 fois plus grand que dans le cas de 9 *ticks*. Cependant, compte tenu de la limite de 64 Go de la mémoire du calculateur, l'exploration n'est pas complète pour 11 *ticks*. Les résultats représentés Figure 6 pour le 11 ticks ont été obtenus sur un ordinateur avec 128 Go de mémoire. Ils nous servent de référence pour mettre en avant l'apport de la décomposition automatique (*splitting*) présentée dans le paragraphe suivant.

Exploration avec partition (splitting)

Pour la vérification des trois propriétés dans le cas de 11 *ticks*, nous avons mis en œuvre la technique de partitionnement automatique qui génère 9 sous-contextes conjointement avec l'algorithme PastFree[ze]. Les sous-contextes ont été composés de façon indépendante avec le modèle. Les résultats de ces explorations sont présentés dans la figure 7. Il convient de noter que, dans ce cas, l'algorithme traditionnel BFS aurait échoué à explorer pour au moins 3 des sous-contextes obtenus (1, 2 et 9). Leur traitement aurait nécessité d'une autre étape de partitionnement, ce qui n'est pas nécessaire en cas d'utilisation du PastFree[ze]. Une autre observation est que, avec la technique de partitionnement automatique, l'espace des états a été décomposé en 9 partitions, ces partitions n'étant pas disjointes. L'exploration globales avec les 9 sous-contextes a

[4] Les résultats bruts présentés dans cette étude ainsi que les fichiers sources sont disponibles sur http://www.obpcdl.org.

parcouru 779 739 813 configurations et 2 611 647 510 transitions, c'est à dire 1,92 fois plus de configurations et 1,93 plus de transitions que dans l'espace des états présenté dans la figure 6. Néanmoins, nous pensons que ce coût est minime au regard du fait qu'il nous permet d'analyser un espace 4,78 fois plus grand sans la nécessité de doubler la mémoire physique de la machine.

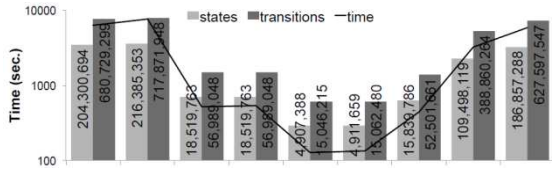

Figure 7 : Résultat pour l'analyse des 9 partitions pour un contexte dans le cas de 11 *ticks*.

5 . Conclusion

Dans cet article, nous avons mis en œuvre une technique de vérification basée sur l'identification de contextes pour l'analyse de trois exigences d'un régulateur de vitesse. L'asynchronisme de ce système rend les approches traditionnelles de *model-checking*, presque impossible. Avec la prise en compte de l'environnement à travers le formalisme CDL, cette tâche devient gérable en s'appuyant sur deux stratégies d'optimisation efficaces. Ces stratégies reposent sur les propriétés structurales des contextes CDL et permettent l'analyse d'accessibilité sur des modèles de tailles supérieures difficilement permise par des approches traditionnelles. L'approche présentée dans cet article offre des résultats prometteurs. Néanmoins, pour que cette technique puisse être utilisée à l'échelle industrielle pour la vérification de propriétés de systèmes critiques, il est nécessaire de proposer au concepteur un cadre méthodologique. Celui doit lui permettre de s'assurer d'une couverture suffisante des contextes spécifiés au regard du comportement du système complet et des propriétés à vérifier.

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, F. Boniol, JC. Roger, L. Leroux : Improving model checking with context modelling. Advances in Software Engineering ID 547157, 13 pages (2012).

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] L. Le Roux, J. Delatour et P. Dhaussy : Modélisation UML d'un régulateur de vitesse automobile; Revue Génie Logiciel, n° 109, Juin 2014.

[16] L. Le Roux, P. Dhaussy et F. Boniol : Vérification formelle de propriétés basée sur une réduction de l'espace d'exploration de modèles; Revue Génie Logiciel, n° 107, décembre 2013.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] C. Teodorov, L. Leroux, P. Dhaussy : Past-free reachability analysis. Reaching further with DAG-directed exhaustive state-space analysis. (submitted to) 29th IEEE/ACM International Conference on Automated Software Engineering (2014).

[22] 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.

Philippe Dhaussy est directeur du pôle STIC de l'ENSTA-Bretagne et enseignant-chercheur HDR. 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.

Luka Leroux est ingénieur de recherche. Après 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.

Ciprian Teodorov est chercheur post-doctorant au laboratoire Lab-STICC UMR CNRS 6285 à l'ENSTA-Bretagne. Ses recherches sont orientées vers l'Ingénierie Dirigées par les Modèles pour l'industrialisation d'outils de preuve pour les systèmes embarqués et systèmes sur puce. Avant intégrer l'équipe IDM du pôle STIC de l'ENSTA-Bretagne il a été ingénieur CAO à Dolphin Integration en charge principalement de l'intégration d'une nouvelle infrastructure de simulation VHDL dans le simulateur mixte SMASH. En 2011 il a obtenu le titre de docteur en informatique de l'Université de Bretagne Occidentale pour ses travaux sur une approche dirigée par les modèles pour la synthèse physique de circuits émergents nanométriques.