Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Extraction de contrats 42 à partir de programmes SystemC

Pierre-Yves Delahaye

Travail d'Etude et de Recherche

Mai 2010

Extraction de contrats 42 à partir de programmes SystemC

Pierre-Yves Delahaye

Grenoble INP - Ensimag

Mai 2010

Résumé

La bibliothèque SystemC de C++ est aujourd'hui de plus en plus utilsée dans le cadre de la conception de haut niveau des systèmes embarqués. Ces systèmes sont constitués de modules qui interagissent entre eux. Afin de pouvoir réutiliser dans d'autres systèmes des modules déjà développés, il est nécessaire de pouvoir en connaître le comportement vis-à-vis de leur environnement. Or, le langage C++ sur lequel repose SystemC ne se prête pas à une extraction directe de modèles comportementaux de ces systèmes. Il est donc nécessaire de transformer les programmes SystemC pour obtenir ces modèles. L'objet du travail réalisé est justement la mise au point d'un outil d'extraction de modèles comportementaux à partir de programmes SystemC.

Mots-clés : SystemC, Contrats 42, Génération automatique

Tuteurs : Kévin Marquet et Matthieu Moy

Table des matières

NOTE AU LECTEUR

Ce rapport a pour objectif de rendre compte du travail qui a été réalisé entre Février et Mai 2010, dans le cadre du module « Travail d'Etude et de Recherche »proposé à l'Ensimag.

J'ai effectué ce travail au sein du laboratoire Verimag de Grenoble. Ce laboratoire travaille sur la mise au point d'outils théoriques et techniques destinés à faciliter le développement de systèmes embarqués sûrs, et ce à des coûts compétitifs.

J'étais encadré par Matthieu Moy, enseignant-chercheur (Ensimag/Verimag), et Kévin Marquet, post-doctorant à Verimag. Je tiens à les remercier pour avoir accepté de suivre mon travail, pour leurs précieux conseils, et le temps qu'ils n'ont pas hésité à consacrer pour m'expliquer le travail à réaliser.

1 Introduction

Les systèmes embarqués développés aujourd'hui sont de plus en plus complexes et représentent des enjeux économiques colossaux, de par le coût de leur conception qui implique de plus en plus d'acteurs, et de par leur niveau de criticité. On comprend donc qu'il est devenu nécessaire à la fois de pouvoir répartir le travail de conception entre plusieurs équipes, ce qui passe par une développement modulaire de ces systèmes, mais également de pouvoir vérifier et valider le travail effectué, aux différents stades de développement, et ce dès les premières phases.

Pour répondre à ce besoin, une bibliothèque de C++ a été mise en place, SystemC. Elle permet effectivement la conception de systèmes sous forme de modules, constitués de processus décrivant leur fonctionnement, ce qui facilite un travail collectif. Cette bibliothèque permet également d'aborder les systèmes à concevoir avec un niveau d'abstraction élevé, c'est-à-dire sans entrer dans les détails de fonctionnement interne de chaque module, mais en se concentrant plutôt sur les éléments qui vont assurer leur synchronisation. Ceci permet donc de valider les systèmes très rapidement, et de pouvoir aborder le développement du logiciel très tôt, sans qu'il y ait besoin d'une partie matérielle définie précisément.

Plusieurs méthodes permettent de vérifier des systèmes, et parmi celles-ci on trouve notamment le model-checking. Or, il se trouve que les outils permettant de mettre en œuvre cette technique de vérification n'acceptent pas les programmes écrits en SystemC, à cause de la complexité du langage C++ sur lequel repose cette bibliothèque. La solution pour résoudre ce problème consiste donc à traduire le code SystemC vers les langages d'entrée des outils de vérification.

En outre, comme nous l'avons dit, il est aujourd'hui très intéressant de pouvoir concevoir un sysème sous forme de modules communiquant entre eux, ce que permet SystemC. Et pour savoir si un module va pouvoir s'intégrer au reste du système, il est nécessaire de pouvoir établir un modèle de son comportement vis-à-vis des autres modules. Ce modèle comportemental peut notamment être représenté par un automate. Dès lors, il peut s'avérer utile d'être capable de générer automatiquement ce type d'automate, à partir de la description SystemC du module auquel on s'intéresse. Or, là encore, du fait de la complexité du langage C++, il est impossible d'effectuer cette génération directement à partir d'un programme SystemC.

On le voit, la mise en place d'une chaîne de transformation de programmes SystemC présente donc un double intérêt : (i) être capable de traduire les programmes SystemC vers les langages d'entrée des outils de vérification, et (ii) être capable d'obtenir un modèle compotemental (par exemple sous forme d'automate) des modules d'un système. Cette transformation de programmes SystemC se passe en deux temps, comme illustré Figure 1.

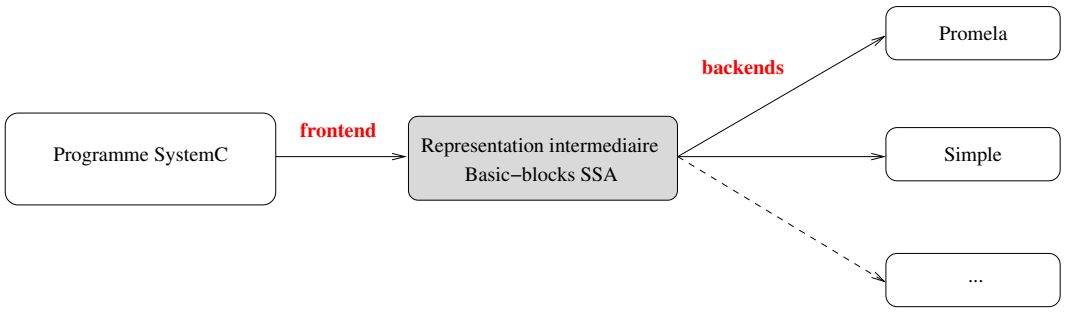

FIGURE 1 – Chaîne de transformation de programmes SystemC

Un première étape consiste à obtenir une représentation intermédiaire du code. Ensuite, on fait appel à un backend qui effectue la transformation du code SystemC, à partir de la représentation intermédiaire, vers la représentation cible choisie (langage d'entrée d'un outil de vérification, automate de description comportementale, ...) : il y aura donc autant de backends que de représentations cibles.

1.1 Aperçu des travaux déjà réalisés dans le domaine

1.1.1 Ce qui était en place avant le TER

Un outil a été développé pour obtenir la forme intermédiaire : il s'agit de PinaVM [6].

On était également capable d'obtenir la représentation du programme SystemC dans le langage Promela, langage d'entrée du model-checker Spin [7].

Enfin, comme nous l'avons vu, il peut être intéressant de modéliser le comportement des modules constituant un système. Pour cela, un nouveau formalisme a été mis en place : il s'agit de 42 [5, 2]. Ce formalisme s'articule autour de la notion de composants, sorte de boîtes noires dotées de ports d'entrée et de sortie, et pouvant correspondre aux modules d'un programme SystemC. Ces composants peuvent ensuite communiquer entre eux, via leurs ports, par l'intermédiaire de signaux. Au-dessus de ces composants, en terme hiérarchique, on trouve le contrôleur : il joue en quelque sorte le rôle de chef d'orchestre, en organisant le fonctionnement des composants réunis. Enfin, le fonctionnement d'un composant, au niveau de ses entrées/sorties, est décrit par un contrat. Un contrat est en fait un graphe, constitué d'états et de transitions. Chaque transition est réalisée en présence d'un certain nombre de signaux sur les ports d'entrée du composant, et à chaque transition sont associés des signaux produits par le composant, et transmis à l'extérieur via ses ports de sortie.

1.1.2 Une des perspectives du travail précédemment effectué, parmi d'autres

Même si, comme nous venons de le dire, une modélisation du comportement des modules d'un système sous forme de contrats 42 a été proposée, il n'y a pour le moment aucun backend capable de construire des contrats 42 à partir d'une description SystemC.

Or, l'assemblage de modules décrits en SystemC nécessite de connaître les spécifications de chacun d'eux par rapport à leur environnement externe. On a ainsi besoin de savoir quand et sous quelles conditions un module va produire tel ou tel signal en sortie, afin de s'assurer que les modules pourront collaborer correctement.

Dès lors, il s'avère intéressant d'essayer d'extraire des contrats 42, à partir de modules définis en SystemC. C'est l'objet du travail relaté par ce présent rapport.

1.2 Contribution du travail réalisé, et structure du rapport

Ayant déjà une représentation intermédiaire du code SystemC, fournie par l'outil PinaVM, il s'agissait de réaliser un backend capable de construire un contrat 42. Il a donc fallu mettre en place des algorithmes capables d'effectuer les traitements appropriés pour la construction des contrats, et ce pour les différentes instructions SystemC qui sont prises en compte dans ces derniers. Nous nous sommes plus particulièrement intéressés à la représentation des attentes et des notifications d'événements, ainsi qu'aux structures de contrôle (`if` et `while`).

La section 2 consistera en un état de l'art des travaux déjà effectués : précisions sur ce qu'est la bibliothèque SystemC, rôle joué par l'outil PinaVM, résultats obtenu grâce au backend de Promela, et définition des contrats 42. Puis, la section 3 précisera le travail effectué, dont ce rapport est l'objet. Cette section décrira notamment les algortithmes mis en place, les difficultés rencontrées, les autres solutions envisageables, et montrera un exemple de résultat fourni par le backend de 42. Enfin, la section 4 sera la conclusion de ce rapport.

2 Contexte du travail, et état de l'art des travaux déjà effectués

2.1 La bibliothèque SystemC et le niveau d'abstraction TLM

2.1.1 Les besoins de l'industrie

La complexité des systèmes embarqués ne cesse de croître. De ce fait, les langages de description matérielle tels que VHDL ne permettent plus de faire des simulations rapides, leur niveau d'abstraction

n'étant pas assez élevé. Afin de pouvoir vérifier très tôt que le matériel développé correspond aux spécifications, dès les premières phases de développement, et ainsi pouvoir commencer rapidement le développement de la partie logiciel des systèmes embarqués, il était donc nécessaire de mettre au point des outils de modélisation autorisant un niveau d'abstraction élevé.

En outre, les temps de développement étant de plus en plus courts, et le travail s'effectuant de plus en plus au sein d'équipes réparties dans des lieux différents, il a fallu mettre au point des outils de conception s'articulant autour de modules, un peu à l'image de ce que font les langages orientés objet. Chaque module représente une fonctionnalité du système, et peut donc non seulement être réutilisé dans plusieurs projets, mais également développé en parallèle avec d'autres modules, réalisés par des équipes différentes.

C'est ainsi qu'a été mis au point un nouveau niveau d'abstraction, appelé TLM (Transaction Level Model). Cette nouvelle approche est basée sur l'utilisation de modules communiquant entre eux. Le niveau d'abstraction élevé permet d'effectuer des simulations rapides, ce qui fait de la modélisation TLM une modélisation très utilisée dans les premières phases de développement.

2.1.2 Comment SystemC répond à ces besoins

Or, il se trouve que la bibliothèque SystemC est compatible avec ce niveau d'abstraction qu'est TLM. Cette bibliothèque présente également l'avantage de posséder un certain nombre de primitives permettant de décrire la synchronisation entre processus. Etant de plus issue du langage C++, elle est donc aujourd'hui très utilisée dans l'industrie des sytèmes embarqués.

Il est donc possible de concevoir en SystemC un système articulé autour de modules, dont le fonctionnement est décrit par un ensemble de threads, et qui communiquent entre eux par l'intermédiaire de ports qui transmettent des signaux (événements, ...). Ces différents modules fonctionnent donc en parallèle, et se synchronisent entre eux notamment grâce aux primitives SystemC `wait` et `notify` qui fonctionnent avec des signaux repésentant des événements. Plus précisément, ces deux primitives agissent ainsi :

- `wait(e:event)` : L'exécution du thread est stoppée, il rend la main à l'ordonnanceur, et l'exécution ne reprendra que lorsqu'un autre thread, et donc un autre module, aura produit l'événement attendu.
- `notify(e:event)` : Cela rend le thread qui attend cet événement éligible par l'ordonnanceur. Le thread en attente pourra donc reprendre son exécution une fois que celui qui a notifié l'événement aura été stoppé.

Notons que nous n'aborderons pas ici les attentes sur du temps (`wait(t:time)`), puisqu'elles ne sont pas encore gérées au niveau des contrats 42.

La possibilité d'utiliser ces primitives est très intéressante, puisqu'elles sont au cœur des concepts de concurrence, essentiels dans la modélisation des systèmes embarqués.

La Figure 2 donne un exemple de deux modules SystemC communiquant entre eux. Lorsque l'on démarre le système, le module1 initialise la variable `a`, et le module2 produit un signal `e1` et incrémente le variable `b`. Puis le module2 se met en attente de `e2`. Le module1 peut alors reprendre la main, en incrémentant `a`, et en produisant le signal `e2`. Le processus `T1` arrive à son terme et meurt. Le module2, qui entre temps a reçu le signal `e2`, peut alors poursuivre son exécution en affectant la valeur 18 à `b`, et il meurt à son tour.

2.2 L'outil PinaVM

2.2.1 De la nécessité de transformer un programme SystemC pour ensuite le vérifier ou modéliser le comportement de ses modules

Un programme SystemC trop complexe pour le vérifier ou en extraire des contrats, sans étape intermédiaire

Les programmes SystemC sont en eux-mêmes difficiles à analyser. En effet, la bibliothèque étant basée sur C++, il n'y a pas de sémantique formelle, et les modèles SystemC sont rendus complexes par une implémentation lourdes d'éléments pourtant relativement simples.

Toutefois, on l'a déjà signalé, il peut être intéressant de vérifier certaines propriétés d'un modèle SystemC, notamment par la technique du model-checking, ou bien d'extraire de ce modèle des spécidications

```
1  void module1::T1(){
2    int a=0;
3    wait(e1);
4    a++;
5    e2.notify();
6  }
7
8  void module2::T2(){
9    int b=0;
10   e1.notify();
11   b++;
12   wait(e2);
13   b=18;
14 }
```

FIGURE 2 – Exemple de deux modules SystemC

comportementales des modules le constituant. Ne pouvant pas appliquer de model-checker ou extraire des contrats directement à partir d'un programme SystemC, étant donné sa complexité, il est donc nécessaire de pouvoir transformer un programme SystemC.

L'importance du format de la représentation intermédiaire

Comme nous l'avons précédemment mentionné, la transformation d'un programme SystemC se fait en deux temps, par l'application d'un frontend qui génère une représentation intermédiaire, puis d'un backend qui applique à la représentation intermédiaire un traitement pour finalement obtenir un code écrit dans le langage de l'outil de vérification choisi, ou obtenir une spécification comportementale (cf. Figure 1).

Or, il existe actuellement une multitude de frontends pour SystemC, mais la plupart ont des limitations, ou sont trop complexes pour être utilisés facilement. En outre, parmi ces frontends, aucun n'est capable de produire une forme intermédiaire SSA (Static Single Assignment), pourtant très populaire, notamment du fait de son efficacité dans les processus de vérification.

2.2.2 Ce que fait PinaVM

C'est ainsi qu'est né PinaVM. Cet outil a pour vocation de surmonter les limitations des frontends développés jusque là, et produit une forme intermédiaire SSA.

Le principe général de PinaVM est décrit dans [6]. Toutefois, donnons quand même quelques éléments clés de son fonctionnement. L'outil est inspiré de Pinapa, un frontend capable de traiter un code C++ (et donc notamment SystemC), et capable d'extraire l'architecture du système modélisé. Cependant, Pinapa est difficile à mettre en œuvre. Pour contourner cette difficulté, l'outil PinaVM fait appel à LLVM, une infrastructure de compilateur. Et l'utilisation de LLVM permet surtout d'obtenir une représentation intermédiaire SSA, forme intermédiaire très bien optimisée pour être ensuite traitée par un outil de vérification.

Sur la Figure 3, on peut voir la forme intermédiaire produite par LLVM (un peu différente de celle obtenue avec PinaVM) pour le thread `T1` du module1 de la Figure 2.

Notons que la représentation de la Figure 3 est un peu différente de la forme intermédiaire obtenue par PinaVM, dans la mesure où cet outil identifie les éléments de communication et de synchronisation entre les différents modules, c'est qui est d'ailleurs essentiel si l'on veut générer des contrats 42.

2.3 Promela et le model-checking

2.3.1 Retour sur une technique de vérification : le model-checking

Parmi les différentes méthodes de vérification de programmes, on trouve le model-checking. Comme mentionné dans l'avant-propos de [1], cette méthode cohabite avec d'autres méthodes, complémentaires, que sont les tests, et les preuves de programme.

```
1  Function : _ZN6thread2T1Ev
2    entry:
3      %"alloca_point" = bitcast i32 0 to i32 ;
4      %0 = getelementptr inbounds %struct.thread* %this, i32 0, i32 1 ;
5      %1 = getelementptr inbounds %struct.thread* %this, i32 0, i32 0 ;
6      call void @_ZN7sc_core9sc_module4waitERKNS_8sc_eventE(%"struct.sc_core::
           sc_module"* %1, %"struct.sc_core::sc_event"* %0)
7      %2 = add nsw i32 0, 1                                    ;
8      %3 = getelementptr inbounds %struct.thread* %this, i32 0, i32 2 ;
9      call void @_ZN7sc_core8sc_event6notifyEv(%"struct.sc_core::sc_event"* %3)
10     br label %return
11
12   return:
13     ret void
```

FIGURE 3 – Exemple de forme intermédiaire générée par LLVM

Les tests, eux, sont empiriques, et ne peuvent être exhaustifs. La preuve manuelle assistée, bien que très efficace, dans la mesure où elle permet de garantir la fiabilité d'un programme, est complexe à mettre en œuvre. Le model-checking est en quelque sorte un intermédiaire entre ces deux méthodes : il s'agit d'une méthode exhaustive, en grande partie automatique. L'ingénieur n'a en effet qu'à construire une modélisation du programme dans le langage reconnu par le model-checker, et à formaliser les propriétés à vérifier.

Le model-checking a déjà permis de mettre en évidence, à l'échelle industrielle, des erreurs bien cachées au sein de systèmes de très grande taille. Il s'agit donc d'une méthode qui a montré son efficacité, et qui continue à être très utilisée notamment dans le domaine des systèmes critiques.

Avant d'analyser un programme, un model-checker construit un graphe constitué d'états et de transitions. Chaque parcours possible du graphe depuis l'état initial représente un cas d'exécution du programme. L'analyse se fait ensuite à partir de ce graphe. Le model-checker indique alors si les propriétés que l'on veut vérifier sont bien respectées.

Une remarque importante s'impose sur la technique du model-checking. Elle ne peut garantir le fonctionnement d'un système, dans la mesure où ce qui est validé est la modélisation du système, et non le système lui-même. Comme tout modèle, le modèle utilisé par le model-checker peut comporter des limites.

2.3.2 Les caractéristiques de Promela et de son model-checker associé : Spin

Comme nous l'avons déjà mentionné, un backend a été conçu pour le langage Promela, langage de modélisation utilisé par le model-checker Spin.

Le système est d'abord décrit via Promela, un langage ressemblant au C, agrémenté de quelques primitives de communication. Avec Promela, on peut donc décrire le comportement des différents modules d'un système, par l'intermédiaire des processus qui définissent leur fonctionnement, et leurs interactions. Une fois cette modélisation effectuée grâce à Promela, le model-checker Spin permet de vérifier, par un parcours exhaustif des différents cas d'utilisation du programme modélisé, que le système vérifie bien certaines propriétés. Une des qualités de Spin est de réduire le nombre d'états du graphe sur lequel repose l'analyse (par rapport à d'autres model-checkers), ce qui est très appréciable quand on sait que le nombre d'états évolue de façon exponentielle avec le nombre de processus du système.

Une description plus détaillée de Promela et de son model-checker associé, Spin, pourra être trouvée dans [1, 3].

On trouvera en annexe (Figure 13) la traduction du thread `T1` du module1 de la Figure 2. Cette traduction n'est donnée que dans le but de faire remarquer au lecteur qu'au niveau des lignes 11 et 15, même la variable `a` interne au processus est prise en compte dans la modélisation Promela, ce qui peut présenter des inconvénients, comme nous allons le voir dans le paragraphe suivant.

2.4 Les contrats 42

2.4.1 Pourquoi ne pas se contenter du backend de Promela

La réutilisation de modules déjà développés, pratique de plus en plus courante, nécessite d'avoir une spécification précise du comportement des modules, afin de s'assurer qu'ils s'intégreront bien au reste du système.

[5] rappelle également qu'il est nécessaire de pouvoir se focaliser sur la synchronisation des modules entre eux, et non sur leur fonctionnement interne, afin de pouvoir vérifier un système dans un temps raisonnable. En effet, prendre en compte l'intégralité d'un programme SystemC, comme le fait Promela, c'est prendre le risque de faire exploser le nombre d'états du graphe dont a besoin le model-checker, ce qui pose bien évidemment des problèmes de temps de calcul pour la machine qui doit vérifier le système. Or, il manque des outils de modélisation qui permettent de faire abstraction des données internes aux modules.

2.4.2 Ce que les contrats 42 apportent de nouveau

Les contrats 42 (cf. [5, 4]) permettent de répondre à ces problèmes, en se focalisant sur la synchronisation des processus, et en s'intéressant plus particulièrement aux entrées/sorties des modules (appelés composants dans le formalisme 42), et à la façon dont ces entrées/sorties sont gérées.

La Figure 4 donne un exemple de composant 42.

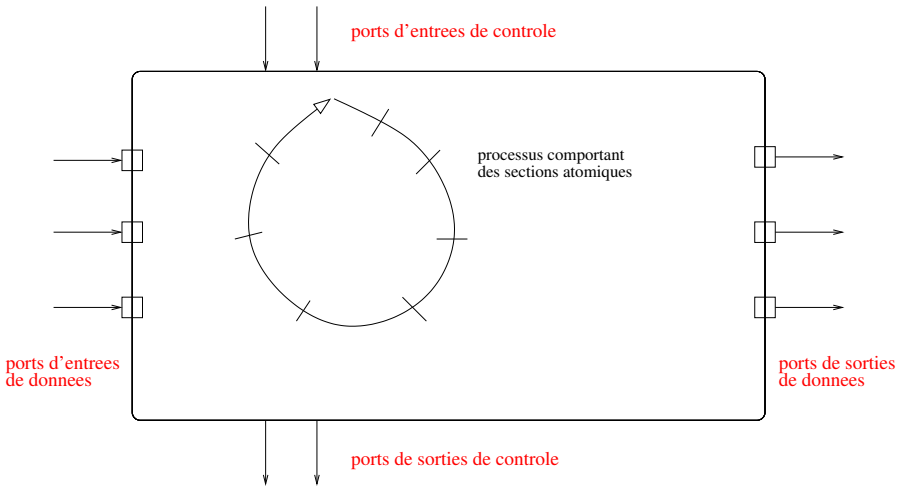

FIGURE 4 – Un exemple de composant 42

On voit donc chaque composant comme une boîte noire, avec des signaux en entrées, des signaux en sorties. Les signaux de contrôle sont ceux associés au contrôleur, dont le rôle est de piloter l'ensemble des composants d'un système. Quant aux signaux de données, ce sont ceux qui assurent la communication avec les autres composants. Le contrat 42 décrit alors les différentes façons dont ces signaux sont gérés par le module (l'ordre dans lequel ils apparaissent, à quel moment et dans quel contexte un signal en entrée est nécessaire, à quel moment et dans quel contexte le module produit un signal en sortie), et cela suffit à s'assurer qu'un module pourra collaborer avec les autres.

La Figure 5 donne l'exemple d'un contrat 42 associé au thread `T1` de la Figure 2.

Un contrat spécifie donc deux choses [2] :

– la séquence d'arrivée des signaux de contrôle qui permettent d'activer le module
– pour chaque activation du module, l'ensemble des informations (i.e. des événements) requis en entrée, et l'ensemble des signaux (signaux de contrôle et de donnée) produits

On remarque qu'à chaque transition est associée une étiquette de la forme :

```
{data req} control input / control outputs {data prod}.
```

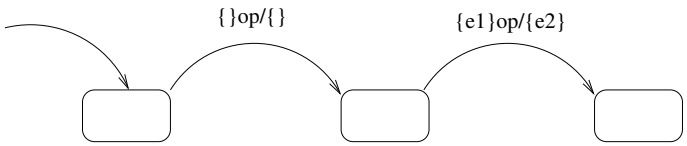

FIGURE 5 – Un exemple de contrat 42

Dans la deuxième transition de la Figure 5, le signal `e1` signifie que la transition ne peut s'effectuer que lorsque l'événement `e1` est présent. L'entrée de contrôle `op` permet d'activer le module (plus précisément le thread qui y est défini), et ainsi d'exécuter l'ensemble des instructions jusqu'au `wait` suivant : cette entrée de contrôle lance l'exécution d'une section atomique. Le signal `e2` produit indique que le module a produit un événement `e2` (correspondant à un `notify`).

3 Extraction de contrats 42 à partir de programmes SystemC

Dans [2], on trouve une définition de la correspondance entre les programmes SystemC et les contrats 42. On nous explique ainsi que chaque composant associé à un thread possède une entrée de contrôle `op` qui permet d'exécuter une section atomique du programme SystemC, depuis le point d'arrêt de la dernière activation, jusqu'au `wait` suivant.

Mais aucun algorihme d'extraction de contrats 42 n'est donné.

Ce que nous allons présenter maintenant, ce sont donc ces algorithmes d'extraction, et nous allons voir que leur mise en œuvre n'est pas sans poser quelques difficultés.

3.1 Le traitement des primitives SystemC `wait(e:event)` et `notify(e:event)`

La Figure 6 montre comment extraire un contrat 42 à partir d'un programme SystemC contenant des `wait` et des `notify`.

3.2 Le traitement des branchements : `if..then..else` et `while`

3.2.1 Un dilemme : une granularité plus fine, ou l'explosion du nombre d'états

Prenons le programme SystemC de la Figure 7. Il correspond au thread caractérisant un module.

Le problème principal qui est apparu durant ce travail est le cas où le programme SystemC comporte plusieurs blocs `if..then..else` consécutifs. Il y a alors deux stratégies possibles pour la génération du contrat. On les présente dans les deux paragraphes suivants :

L'explosion du nombre d'états

Le contrat 42 représenté Figure 8 fait exploser le nombre d'états du graphe.

L'état `A` correspond au dernier état créé avant la succession de `if..then..else`. De cet état, on fait partir une branche pour chaque parcours possible de la succession de blocs `if..then..else`. Dans chaque branche, on trouvera donc autant d'états qu'il y a de primitives `wait(e:event)` dans les blocs `if/else` parcourus.

Faisons maintenant un peu de dénombrement pour évaluer le nombre d'états générés dans le contrat 42.

Si nous avons n blocs `if..then..else` consécutifs, nous aurons alors 2^n branches qui partiront de l'état `A`. Ainsi, chaque bloc `if/else` sera représenté 2^{n-1} dans le contrat généré. Or, l'algorithme représenté Figure 6 montre que l'on crée un état chaque fois que l'on rencontre un `wait(e:event)`. Soit n_{wait} le nombre total de structures `wait(e:event)` des blocs `if/else`. Nous pouvons donc dire qu'entre les états `A` et `B`, nous aurons généré $n_{explosion}$ états, avec :

$$n_{explosion} = n_{wait} 2^{n-1} \quad (1)$$

```
-- Quand on detecte un WaitEvent :

   - S'il existe des evenements attendus ou notifies
     (existNotify ou existWait) :
     => on cree un nouvel etat
     => on met en place une transition entre cet
        etat et le dernier etat precedemment cree;
        les entrees de donnees correspondent aux derniers
        evenements attendus releves, et
        les sorties de donnees correspondent aux derniers
        evenements notifies
     => on met a jour la variable qui identifie
        le dernier etat cree
     => on vide la liste des evenements attendus et
        notifies, releves depuis la derniere creation d'un
        etat
     => on ajoute dans la liste des evenements attendus
        l'evenement associé au WaitEvent qui vient d'etre
        detecte
     => on met la variable existNotify a false
     => on met la variable existWait a true

   - Sinon, s'il n'y a pas d'evenements attendus ou notifies :
     => on ajoute dans la liste des evenements attendus
        l'evenement associé au WaitEvent qui vient d'etre
        detecte
     => on met la variable existWait a true


-- Quand on detecte un Notify :

   => on ajoute dans la liste des evenements notifies
      l'evennement associe au Notify qui vient d'etre
      detecte
   => on met la variable existNotify a true
```

FIGURE 6 – Algorithme de traitement des `wait` et `notify`

Nous pouvons donc dire que le nombre d'états générés évolue de façon exponentielle. Ceci est gênant, dans la mesure où, comme nous l'avons déjà dit, les contrats 42 ont justement été mis en place pour essayer de modéliser simplement un système, en réduisant au maximum le nombre d'états du graphe qui le modélise. En revanche, l'intérêt de cette stratégie que nous venons d'exposer est que la granularité du programme SystemC est respectée. Nous ne créons effectivement de nouveaux états qu'en présence d'un `wait(e:event)`.

Une granularité plus fine

Au niveau de l'automate qui représente le contrat 42, il faut savoir que chaque transition correspond à une section atomique du code du processus modélisé. Ainsi, pour deux automates modélisant le même processus, si l'un comporte plus de transitions que l'autre, on peut dire qu'il comporte plus de sections atomiques différentes, et donc que la granularité est plus fine.

Le contrat 42 représenté Figure 9 augmente la granularité, par rapport au programme SystemC.

```
1  void module1::T1(){
2    int a=0;
3    int b=0;
4
5    ...
6
7    if(a==0){
8      ...        // IF1
9    }
10   else{
11     ...        // ELSE1
12   }
13
14   if(b==0){
15     ...        // IF2
16   }
17   else{
18     ...        // ELSE2
19   }
20
21   ...
22
23 }
```

FIGURE 7 – Structure de programme SystemC pouvant conduire à une explosion du nombre d'états

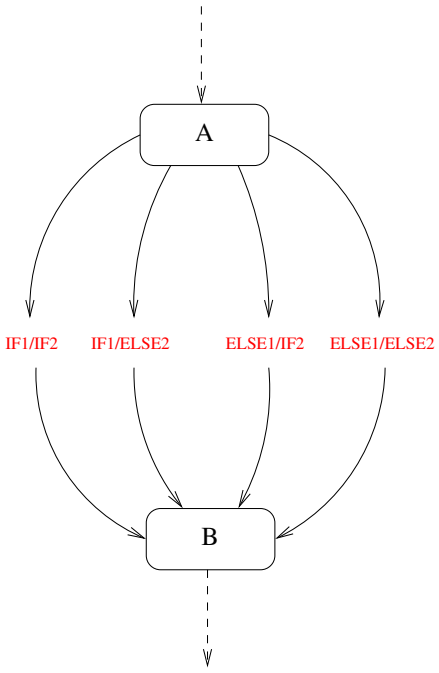

FIGURE 8 – Une stratégie de génération de contrat 42 : explosion du nombre d'états

Ici, la granularité est plus fine que celle du programme SystemC que l'on cherche à modéliser. En effet, nous créons artificiellement un état `B`, au niveau duquel les branches `IF1` et `ELSE1` se rejoignent. En revanche, le nombre d'états générés, que nous appelerons $n_{granularite}$, est :

$$n_{granularite} = n + n_{wait} \tag{2}$$

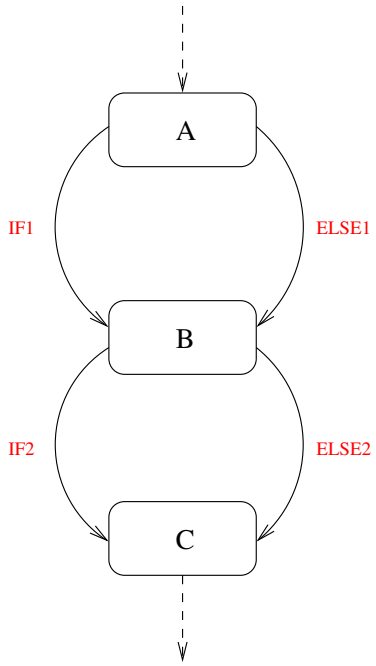

FIGURE 9 – Une stratégie de génération de contrat 42 : augmentation de la granularité

3.2.2 La solution implémentée

En comparant le nombre d'états générés par chacune des stratégies (cf. Equations 1 et 2), on voit immédiatement que la stratégie de la Figure 9 génère beaucoup moins d'états.

Or, la réduction du nombre d'états générés est quand même l'un des aspects fondamentaux des contrats 42. Donc, même si avec cette stratégie, la granularité augmente, nous avons décidé d'implémenter celle-ci, plutôt que celle qui fait exploser le nombre d'états.

La Figure 10 correspond à l'algorithme de traitement des branchements. Comme on peut le constater, il ne fait que la distinction entre les branchements conditionnels, et les autres. Le traitement d'un branchement correspondant à un `if..then..else`, à un `while`, ou à une boucle `for` est effectivement le même.

3.3 Un exemple d'extraction de contrat 42 avec l'outil développé

La Figure 11 donne une exemple de programme SystemC que nous sommes capables de traiter avec le travail réalisé.

Le contrat généré par l'outil est alors celui de la Figure 12.

On obtient donc un contrat 42 sous forme de fichier `xml`, dans lequel l'événement `e1` a été codé sous la forme `toto0xbfbfd5a8`, et l'événement `e2` sous la forme `toto0xbfbfd5e80`.

```
- Si le Basic-Block sur lequel on se branche a déjà été parcouru :
  On réalise une transition vers le premier état de ce Basic-Block

- Sinon :
  - Si on a un branchement conditionnel :
    => on parcourt les instructions correspondant au cas où
       la condition est vraie
    => puis, on parcourt les instructions correspondant au cas où
       la condition est fausse

  - Sinon :
    => On crée un nouvel état
    => On marque le Basic-Block sur lequel on se branche
    => On parcourt ce Basic-Block
```

FIGURE 10 – Algorithme de traitement des branchements

```
1  void module1::T1(){
2      int a=0;
3      if(a<4){
4        e1.notify();
5        while(a<8){
6   wait(e2);
7   a++;
8        }
9        wait(e1);
10       a++;
11     }
12   }
13 }
```

FIGURE 11 – Exemple de programme SystemC

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <!DOCTYPE contract PUBLIC "toto-CT" "contract.dtd">
3  <contract interface="module1-ITF" name="module1-CT">
4    <initials>
5      <state name='f0'/>
6    </initials>
7    <transitions>
8      <transition lab='{}op/{toto_0xbfbfd5a8;} src='f0' sink='f1' />
9  ____<transition lab='{}op/{} src='f0' sink='f3' />
10     <transition lab='{toto_0xbfbfd5e8_0}op/{} src='f1' sink='f1' />
11 ____<transition lab='{}op/{} src='f1' sink='f2' />
12     <transition lab='{toto_0xbfbfd5a8_0;}op/{} src='f2' sink='f3' />
13 __</transitions>
14 </contract>
```

FIGURE 12 – Contrat 42 généré à partir du programme de la Figure 11

4 Conclusion

Nous avons donc développé une ébauche de backend pour les contrat 42. Ce backend est capable de traiter les primitives SystemC `wait(e:event)` et `notify(e:event)`, ainsi que les structures `if..then..else`, `while`, et les boucles `for`.

Les contrats 42 font également apparaître des appels de fonctions d'un module à un autre. Il s'agit d'un point important, puisque ces appels de fonctions sont en partie responsables des interactions qu'il y a entre les différents modules d'un système. Or, le backend développé ne gère pas ces appels de fonction. Afin de développer l'outil, il faudrait donc probablement commencer par mettre en place la gestion de ces appels de fonction.

D'autres perspectives pourraient être de vérifier que les contrats générés repectent bien la définition des contrats 42, et surtout de voir si la stratégie choisie de l'augmentation de la granularité est judicieuse.

Enfin, on pourrait essayer de développer un outil permettant de vérifier que différents contrats sont compatibles, ce qui permettrait de s'assurer que les composants relatifs à ces contrats pourront se synchroniser correctement.

ANNEXE

```
1  /*----- Threads -----*/
2
3  proctype _ZN6thread2T1Ev_pnumber_0()
4  {
5  atomic {
6  int llvm_cbe_alloca_20_point;
7      int llvm_cbe_tmp__1;
8
9      skip;
10     /*OutputLValue*/
11     llvm_cbe_alloca_20_point = 0;
12     wait_toto_0xbfaf7378_0();
13 ;
14     /*OutputLValue*/
15     llvm_cbe_tmp__1 = 0 + 1};
16     notify_toto_0xbfaf73b8(0);
17 ;
18 finished[0] = true;
19 }
20 }
21
22 /* ---------- Init ---------- */
23 init
24 {
25     int i = 0;
26     do
27       :: i == NBTHREADS -> break;
28       :: else ->
29          e[i] = 0;
30          T[i] = 0;
31          finished[i] = false;
32          i++;
33     od;
34
35
36     atomic {
37         run _ZN6thread2T1Ev_pnumber_0();
38     }
39 }
```

FIGURE 13 – Exemple de code Promela généré à partir d'un programme SystemC

Références

[1] Béatrice Berard, Michel Bidoit, François Laroussinie, Antoine Petit, and Philippe Schnoebelen. *Vérification de logiciels : Techniques et outils du model-checking*. Vuibert, 1999. 2.3.1, 2.3.2

[2] Tayeb Bouhadiba, Florence Maraninchi, and Giovanni Funchal. Formal and executable contracts for transaction-level modeling in systemc. Technical Report TR-2009-7, Verimag Research Report, 2009. 1.1.1, 2.4.2, 3

[3] Bell Labs. Spin : general tool description. http ://spinroot.com/spin/whatispin.html. 2.3.2

[4] Florence Maraninchi and Tayeb Bouhadiba. 42 : Programmable models of computation for a component-based approach to heterogeneous embedded systems. 2007. 2.4.2

[5] Florence Maraninchi and Tayeb Bouhadiba. 42 : Programmable models of computation for the component-based virtual prototyping of heterogeneous embedded systems. Technical Report TR-2009-1, Verimag Research Report, 2009. 1.1.1, 2.4.1, 2.4.2

[6] Kevin Marquet and Matthieu Moy. Pinavm : a systemc front-end based on an executable intermediate representation. Technical Report TR-2010-8, Verimag Research Report, 2010. 1.1.1, 2.2.2

[7] Kevin Marquet, Matthieu Moy, and Bertrand Jeannet. An asynchronous semantics of systemc in promela. Technical Report TR-2010-7, Verimag Research Report, 2010. 1.1.1