Modélisation conjointe logicielle-matérielle assistée par model-checking

Jean-Christophe Le Lann
Philippe Dhaussy
Lab-STICC UMR CNRS 6285
ENSTA-Bretagne
29806 BREST cedex 9
{jean-christophe.le_lann, philippe.dhaussy}
@ensta-bretagne.fr

Pierre-Laurent Lagalaye
Modaë Technologies
Rennes Atalante Beaulieu
35708 RENNES CEDEX 7
pierre-laurent.lagalaye@modae-tech.com

ABSTRACT

L'ingénierie des architectures logicielles basée sur le prototypage rapide conduit à des itérations nombreuses dont le coût en simulation ne peut être négligé. Ceci se révèle particulièrement crucial pour les applications multimédia (encodeurs, décodeurs vidéo, etc), qui cumulent des spécificités de flux de données, de contrôle et de traitements complexes, découpés en automates à grains fins. Les outils de modélisation HW/SW de telles applications souffrent donc d'une forme d'incompatibilité entre les ambitions en terme de prototypage et les besoins en tests incessants, consommateurs de temps. L'utilisation des techniques de vérification formelle par "model-checking" constitue une solution à cette incompatibilité notoire. Nous décrivons dans cet article un travail exploratoire qui étudie l'association d'un outillage original de modélisation d'architectures mixtes logiciel-matériel basé sur le langage Ruby avec un outil de vérification, nommé OBP (Observer-Based Prover), basé sur les observateurs. Nous illustrons notre contribution par un exemple et décrivons des résultats sur les preuves d'exigences menées.

Keywords

Architectures de calcul, Performances de calcul, Vérification formelle, Patrons de propriétés, Observateurs.

1. INTRODUCTION

Le prototypage rapide d'applications concurrentes permet à l'ingénieur numéricien d'obtenir une première version simplifiée d'un système virtuel et d'engager un processus de concertation sur les caractéristiques que devra présenter ce système. Cette méthodologie agile prone des itérations courtes et un dialogue permanent entre le concepteur et le client, et s'éloigne des processus dits "cérémoniels" où le donneur d'ordre est sensé posséder un lot d'exigences parfaitement formulées dès le démarrage du projet. A terme, l'ingénieur passe du prototype à un produit fini, par une suite de rafinements conjoints des exigences et du système qui les supporte. Toutefois, cette méthodologie pose plusieurs difficultés, parmi lesquelles un juste partage du temps d'ingénierie entre l'écriture des modèles applicatifs et leur validation. L'ambition initiale d'itérer par cycles courts est généralement entravée à la fois par les temps de simulation, et la définition-même des jeux de test propres à activer les fonctionnalités du modèle concernées par l'itération. Il parait judicieux de chercher à coupler la notion de prototypage rapide aux avancées des techniques formelles de vérification.

Cet article présente une expérimentation du couplage d'un outil de modélisation système et d'un outil de vérification formelle de propriétés : l'outil de modélisation d'applications concurrentes et de synthèse de systèmes est développé par la société Modaë Technologies. L'outillage de vérification formelle est constitué de l'outil OBP-Explorer, qui accepte en entrée le langage formel Fiacre. Nous cherchons à renforcer la chaîne de modélisation par une assistance du model-checking. Cette démarche est explicitée sur la figure 5. Il est intéressant de noter que deux utilisations du model-checking sont présentées : la vérification formelle de comportements concurrents, mais également l'analyse de performances du système.

Nous commençons en partie 2 par recenser des travaux similaires à notre approche.

La partie 3 présente ensuite un exemple illustratif d'architecture de calcul pour la présentation des concepts développés. En partie 4, nous présentons l'outillage original basé sur un DSL interne, écrit en langage Ruby, permettant la génération d'architectures mixtes matérielles-logicielles. La partie 6 décrit la technique de vérification formelle et l'outil OBP. L'expérimentation est présentée en partie 7.2 ainsi que la méthodologie utilisée. Nous donnons des résultats sur les preuves menées. En partie 8, nous discutons de notre approche, en tirons un premier bilan et des perspectives pour la poursuite de nos travaux.

Nous concluons en partie 9.

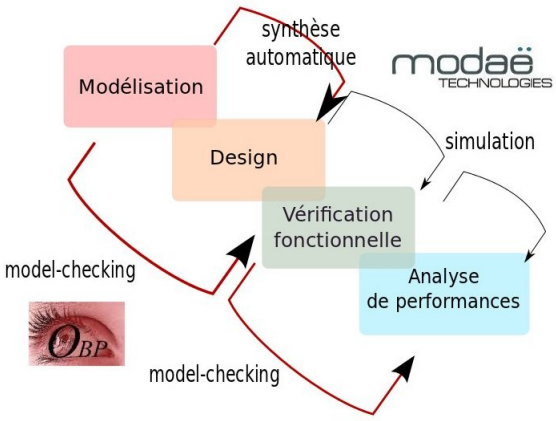

Figure 1: Utilisation d'un model-checker dans une flot de modélisation, sujet de l'étude.

2. ETAT DE L'ART

2.1 Les trois axes du codesign HW/SW

Le domaine de l'EDA (Electronic Design Automation) est un domaine extrêmement actif et productif en terme de recherche académique et industrielle. Alors cantonné à la production correcte de circuits à bas niveau (synthèse logique, dimensionnement des transistors, layout, placement-routage, caractérisation,...), les années 90 ont vu une première tentative d'évolution notoire des niveaux d'abstraction adressés par ses outils [29] : plusieurs outils et méthodologies associées ont tenté de fournir une aide à la conception lors des phases amont de spécification fonctionnelle des systèmes. Ainsi l'outil de cosimulation Cossap [24] peut-être vu comme un ancêtre de bon nombre d'outils : il s'agit de fournir un moyen d'expliciter une découpe de l'algorithme, et de simuler le comportement d'ensemble de blocs interconnectés. Cette première vague d'outils avancés n'a pas recu le succès escompté. Les barrières à cette adoption sont exposées dans [16]. Il a fallu attendre les années 2000, pour voir l'EDA se scinder en deux parties, faisant émerger l'ESL (Electronic System Level) comme discipline à part entière, addressant précisemment les phases amont de la conception : spécification orientée logicel (par exemple SystemC) , modélisation et synthèse HLS [6] de tout ou partie d'un System-on-chip, vérification par simulation [26].

Cette émergence et l'évolution des outils vont désormais de paire avec la complexité des applications : on estime qu'il existe un facteur 6 entre la complexité d'un décodeur MPEG2 (1998) et celle d'un décodeur MPEG4 (AVC, 2005) : si un seul ingénieur était en mesure de comprendre l'intégralité de la réalisation d'un tel décodeur MPEG2, cela est probablement impossible pour les détails de la version MPEG4, aux subtilités reconnues, inhérente aux algorithmes des différentes parties de la norme. L'ESL entend donc apporter des outils adéquats vis-à-vis de cette complexité croissante. Le consortium ITRS (International Technol- ogy Roadmap for Semiconductors) a d'ailleurs désormais clairement identifié [23] que les outils, langages et méthodologies de modélisation exécutable à haut niveau (HLM "high level model") comme un facteur de succès indispensable à la réalisation des futurs systèmes électroniques mixtes logiciel-matériel.

Parmi les travaux remarquables dans le domaine de la capture applicative à haut niveau, un classement selon trois axes peut être proposé :

- **Proposition de nouveaux langages**, au nom évocateur (SystemC, SpecC, HandelC, SystemVerilog, SysML...)
- **Analyse et formalisation des modèles de calculs** (MoC), formalisant des styles de communications rencontrés dans différents types d'application : communication par rendez-vous synchrone, fifos bornées ou non (KPN), event-driven, réactif-synchrone, ...
- **Synthèse automatique au niveau système**, permettant de compiler un comportement séquentiel classique (code C adapté, SystemC,...) en un dispositif matériel.

Parmi ces trois axes, le second est essentiel pour la suite du présent article : les MoCs constituent autant de protocoles abtraits et parfaitement explicites qui assurent la synchronisation des échanges. La maîtrise de cette sémantique doit à terme autoriser l'utilisation de model-checkers dans une chaîne de modélisation a priori non-conçue pour cela : en ce, les MoCs constituent une opportunité que nous tentons d'explorer ici.

Ces trois axes mentionnés au dessus sont malheureusement jusqu'ici relativement cloisonnés : par exemple, l'étude de l'association de modèles de calculs précis à des langages a été délaissée, reproduisant par ailleurs les errements sémantiques des langages informatiques classiques. Les concepteurs des langages phare de l'ESL ont donc rarement associé une sémantique précise et explicite aux capacités de communication fournie par leur langage. Cet état de fait a comme conséquence importante un grande difficulté à introduire les méthodes formelles dans le domaine des systèmes embarqués, de manière convaincante, à un niveau d'exigence industrielle. La présente étude peut être vue également comme une nouvelles tentative en ce sens.

2.2 Les tentatives de formalisation des langages de l'ESL

Nous tentons ici de résumer les exceptions importantes à l'écueil précédent. Dès les années 80, les concepteurs des langages synchrones (Esterel, Lustre, Signal) ont résolument mené une réflexion simultanée sur les deux axes langages concurrent/sémantique formelle [1]. Toutefois, aucun des langages n'a pu s'imposer de manière évidente, par ailleurs rattrapées par la complexité des nouveaux systèmes à concevoir : ainsi, Scade, qui s'appuie sur Lustre, propose des primitives cohérentes, mais d'un niveau tel qu'il n'est pas pensable de chercher à modéliser la plupart des fonctionnalités des SoCs [2]. HandelC, commercialisé et utilisé de manière effective dans la modélisation par processus de type CSP et la synthèse ESL, continue de susciter des tentatives de formalisation [4].

Dans [31] Vercauteren et al. propose un calcul de processus comme modèle intermédiaire qui assure la jonction entre les constructions d'un langage orienté processus communicants et l'implémentation concrète des modèles décrits. Un schéma de traduction est proposé entre le langage et cette représentation, fondée sur les réseaux de Petri.

Un effort conséquent a été conduit par plusieurs chercheurs en matière de codesign, dans un but de génération de code séquentiel linéarisé de manière quasi-statique, c'est-à-dire qui minimise les besoins en décisions d'un ordonnanceur dynamique, à des fins de génération de simulateur ou de runtime. Ce travail est excellemment comparé dans [10]. Dans [31] un calcul de processus est proposé comme modèle intermédiaire qui assure la jonction entre les constructions d'un langage orienté processus communicants et l'implémentation concrète des modèles décrits. Un schéma de traduction est proposé entre le langage et cette représentation, fondée sur les réseaux de Petri.

Des travaux similaires marquants sont à noter [11],[13],[19] – tous autour de la notion d'acteurs [22] – dont Ptolemy ([12]) peut être vu comme le pionner en matière de codesign hw/sw. Les connexions entre le modèle par acteur et les diagrammes SysML et UML sont discutées par Lee dans [25].

Dans l'ensemble de ces travaux que nous venons de citer, la formalisation des sémantiques de communication y est une aide précieuse à cette fin, sans toutefois ambitionner une véritable mise en oeuvre de model-checking *complet* de l'interaction comportement-communication, comme pouvant accompagner le flot de conception lui-même, comme adjonction aux flots de simulation classiques. Ce type de méthodologies est par contre appproché par [7] : dans Metropolis les auteurs proposent en effet de limiter le comportement des processus communicants par des contraintes LTL (Linear Temporal Logic), qui viennent renforcer la spécification du système. Les auteurs, qui attaquent le problème de la détection de deadlock par simple simulation, ne vont pas jusqu'au bout de l'exercice qui consisterait précisément à utiliser ces formules pour enchaîner un véritable model-checking.

2.3 Assistance du flot ESL par model-checking

En ce qui concerne le domaine de la vérification formelle, un certain nombre d'outils font depuis longtemps partie de la panoplie du concepteur de SoCs . Ainsi l'outil Formality de la société Synopsys réalise des preuves d'équivalence fonctionnelle entre différentes netlists. Toutefois le domaine du model-checking reste encore un sujet de recherche actif. La disponibilité du langage PSL standard de-facto de l'ESL pour l'expression de propriétés temporelles (utilisé comme langage d'assertion pour la simulation), autorise des premières expériences de vérification formelle dans l'ESL : une bonne introduction peut être trouvée dans [16] et [30]. [17] a proposé une vérification formelle, par techniques de bounded model-checking [3], de modèles SystemC non-temporisés au niveau transactionnel (c'est-à-dire antérieure au rafinement RTL). L'approche, convaincante, se heurte toutefois à la modélisation de l'ordonnanceur de SystemC lui-même : le modèle de calcul par événements discrets de ce type de simulateur doit être capturé, mais nous considérons que son caractère centralisé peut-être vu comme un obstacle à la généralisation de l'approche pour les phases aval, à savoir la synthèse automatique, qui assure quant à elle l'exécution distribuée (spatiale) des processus. Le recours que nous avons à des MoCs synchrones ou asynchrones, fonctionnant respectivement par rendez-vous et par FIFO, assurent les deux aspects de la vérification : vérification des propriétés et préservation lors de la synthèse. Un travail similaire est présenté dans le framework Kratos [21], et se repose sur Uppaal.

3. UN CAS D'ÉTUDE ILLUSTRATIF

Le cas d'étude retenu présente trois processus communicants (figure 2) qui échangent des données par canaux synchrones et asynchrones. Les calculs arithmétiques réalisés par ces processus restent simples : le premier processus incrémente une variable interne, l'élève au carré et envoie cette donnée sur un port de sortie. Le second processus reçoit la donnée et en extrait une racine carrée, par un algorithme itératif, pouvant prendre un temps dépendant de la valeur de la donnée à traiter. Le résultat est envoyé vers un troisième processus réceptacle. Notons toutefois, dans les phases amont de la modélisation, l'outillage Modaë ne préconise pas le recours à un formalisme d'entrée sous forme d'automates : comme nous le verrons plus bas, ces automates sont au contraire le résultat d'une compilation vers du matériel, et servent également à la génération de code Fiacre, entrée du model-checker.

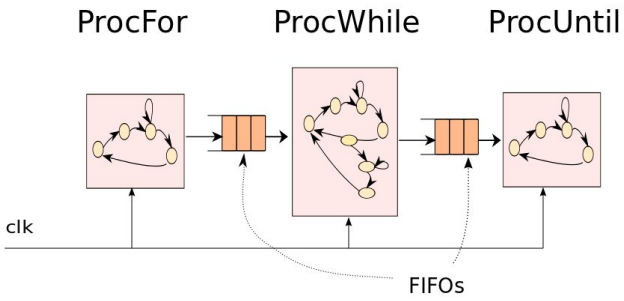

Figure 2: Le circuit de calcul considéré.

Bien que simple, ce modèle est intéressant, car il permet d'exhiber des comportements dépendants de l'architecture générale du système généré par le flot ESL, notamment : vitesses relatives de production et de consommation, sensibilité aux tailles des FIFOs. Nous insistons sur le fait que dans notre exemple choisi, ces comportements fluctuent en fonction de la valeur des données elles-mêmes.

Nous cherchons donc à explorer ces comportements, afin d'aider le concepteur à rapidement appréhender la pertinence de son modèle applicatif, ainsi que l'architecture système retenue. Parmi les questionnement du concepteur, on tente de répondre à des questions typiques :

- Quand les FIFOs sont-elles pleines ?

- Sont-elles bien dimensionnées mutuellement pour qu'un maximum de processus ne soient pas bloqués ?
- Partant de tailles de FIFO imposées, un composant X calcule t il assez rapidement ?
- Existe-t'il des deadlocks possibles dans certains cas de traitement ?

4. UN OUTILLAGE ESL ORIGINAL

Deux outils sont utilisés dans le cadre de l'étude. Un premier outil est l'outil Modaë Technologies qui permet la capture applicative à haut niveau : Le défi de la modélisation en matière de co-design a conduit Modaë Technologies a développer un framework de simulation et de synthèse de systèmes.

La méthodologie associée prône un découpage explicite des algorithmes complexes sous forme de blocs diagrammes, communicants par ports et canaux annotés synchrones ou asynchrones. Dans la version étudiée, le comportement de chaque bloc est écrit dans le langage de script Ruby, typé dynamiquement. La chaîne Modaë de synthèse système (figure 3) transforme cette entrée en un réseau d'automates communicants, à granularité très fine, compatible avec les exigences de la synthèse matérielle. Une de ses caractéritiques remarquables est sa capacité à inférer les types de données, non explicités dans le code Ruby. Le code synthétisé final délivré est constitué de blocs VHDL de niveau transfert de registres (RTL) et de blocs de code C compilable sur processeur embarqué, accompagné du firmware permettant la communication entre VHDL et C embarqué. Le code C contient potentiellement un ordonnanceur dès lors que plusieurs blocs algorithmiques ont été assigné au même processeur. Un parallélisme massif de traitement est bien entendu souhaitable sur FPGA, afin de tirer partie des avantages de telles architectures. Le schéma 4 en donne le principe : à gauche se trouve un processeur RISC et à droite un circuit reconfigurable de type FPGA. Un bus assure la communication entre les deux (la génération du firmware logiciel, ainsi que de l'interface registre, mappée en mémoire, est assurée par le compilateur Modaë). Sur cette figure, le FPGA embarque les processus logiciel synthétisés par le même compilateur. Des FIFOs assurent l'asynchronisme de traitement entre ces processus. Notons que l'outil est également capable de compiler ces processus vers le processeur RISC : on retrouve là l'ambition de la modélisation système amont, qui ne fige aucun choix d'implémentation a priori.

Le deuxième outil est OBP-Explorer[1] [8]. OBP (Observer-based Prover) est un explorateur de modèles permettant de vérifier des propriétés sur l'ensemble de leurs comportements. OBP prend en entrée des modèles décrits en langage CDL (Context Description Language) [9] permettant d'exprimer des scénarii et des propriétés. Les scénarii permettent de reduire l'espace des états lors de l'exploration des modèles. Nous n'aborderons pas cet aspect dans cet article. OBP accepte également en entrée le langage Fiacre [14], qui sert de support à la modélisation du système étudié. Fiacre possède une sémantique précise d'automates communicants de manière synchrone et asynchrone.

[1]Développé à l'Ensta Bretagne http://www.obpcdl.org.

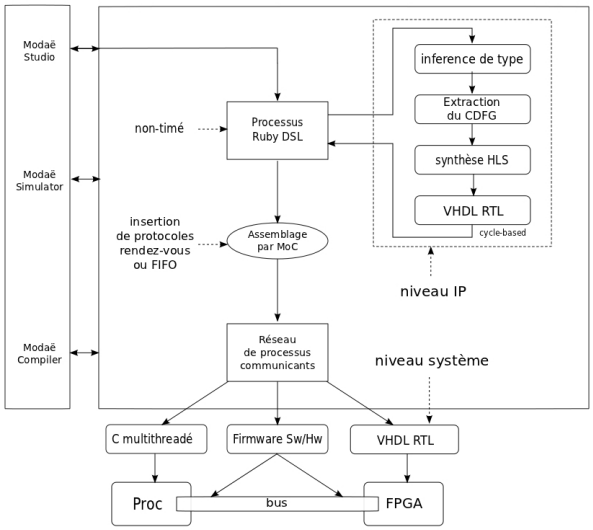

Figure 3: Le flot ESL de modélisation et génération de code selon Modaë Technologies

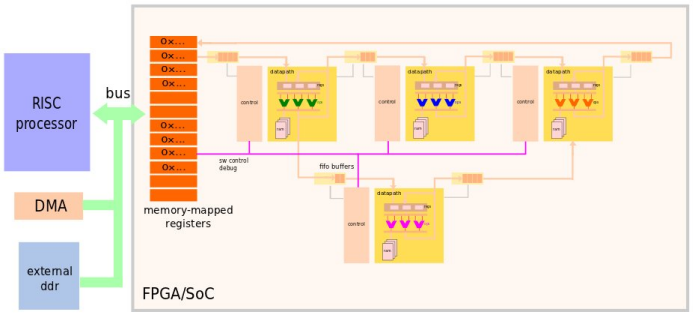

Figure 4: Architecture mixte logicielle-matérielle ciblée par le compilateur Modaë dans cet article.

La chaîne d'outils utilisée ici peut s'illustrer sur la figure 5 : notre travail commence par l'utilisation d'un outil de capture système, écrite dans un DSL Ruby, présentant un asynchronisme fort entre blocs, présent dans le modèle de calcul. La seconde étape consiste à générer un code VHDL qui explicite les comportements des processus sous forme d'automates, synchronisés sur une horloge globale. Ce code respecte la topologie et l'asynchronisme de comportement par insertion de FIFOs matérielles. Les automates ainsi exhibés sont un bon formalisme pour la vérification, que nous enchaînons dans une troisième étape ; au préalable le langage CDL est utilisé afin de formaliser un certain nombre de propriétés souhaitées sur le système généré.

5. PRINCIPES DE TRANSFORMATION D'UN MODÈLE RUBY

Comme représenté sur la figure 5, nous nous intéressons à plusieurs niveaux de représentations peuvent être évo-

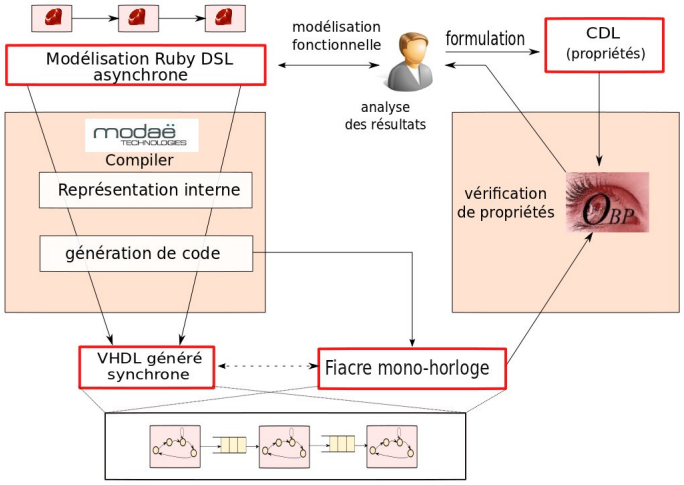

Figure 5: Le flot ESL de modélisation et de vérification associé

quées : la première, la plus intuitive et la plus pratique pour l'architecte, est dite "*untimed*" (dans la terminologie ESL) ou "asynchrone" (dans la terminologie de vérification formelle), dans le sens où aucune horloge n'a été utilisée dans la description du comportement. La dernière est au contraire le résultat d'une compilation (synthèse comportementale) qui a pour résultat un circuit synchrone mono-horloge : tous les éléments de mémorisation (variables, mémoires, FIFO, arbitres,...) sont cadencés par un *tick* commun (horloge physique périodique). Ce choix n'est pas arbitraire, mais est conforme aux recommandations en vigueur en matière de VLSI, qui requiert autant que possible une réalisation synchrone (évitemment de situations de métastabilité [15]). Le présent article s'intéresse précisément à ce modèle.

Les automates matériels ont été établis à partir d'une première compilation du DSL initial Ruby en CDFG (control-data flow graphs) : c'est le procédé de synthèse comportementale classique [18]. Dans la version étudiée ici, on se limite à l'extraction du graphe de flot de contrôle (CFG), qui permet d'obtenir une première version des automates VHDL communicants synthétisables sur FPGA ou ASIC, et que l'on identifie à des automates Fiacre dont on maîtrise l'équivalence sémantique. Une version plus complète consiste (non-traitée dans cette article) à identifier également des états supplémentaires, qui restituent la notion d'ordonnancement des DFG (data-flow graphs). A titre d'exemple, notre extraction des automates à partir du modèle initial Ruby peut donc se résumer à la figure 6. A partir du CFG, il est aisé d'identifier les états de l'automate matériel : ils sont nécessaires dès qu'une boucle de traitement est découverte (les boucles combinatoires sont proscites dans les flots de conception électroniques).

```
class ProcWhile < Reactiv
  inports :i1
  outports :o1

  def behavior
    while true
      i = receive(:i1)
      puts "ProcWhile is computing sqrt of #{i}"
      s = 1
      r = 3
      while s <= i
        s += r
        r += 2
      end
      r = r/2 - 1
      puts " => result = #{r}"
      send(r, :o1)
    end
  end
end
```

Listing 1: exemple de code Ruby-DSL saisi par l'utilisateur.

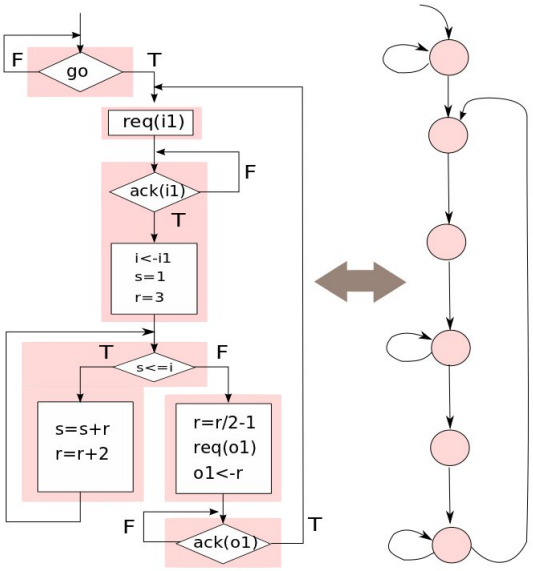

Figure 6: Elaboration d'un automate matériel à partir de l'exemple Ruby précédent. Cet automate, également généré en Fiacre, sert également d'entrée au model-checker.

6. LA TECHNIQUE DE VÉRIFICATION OBP

6.1 Principes de vérification des propriétés

Parmi les techniques de vérification de propriétés sur des modèles logiciels, les méthodes formelles ont contribué, depuis plusieurs années, à l'apport de solutions rigoureuses et puissantes pour aider les concepteurs à produire des systèmes non défaillants. Dans ce domaine, les techniques de model-checking [28, 5] ont été fortement popularisées grâce à leur faculté d'exécuter automatiquement des preuves de propriétés sur des modèles logiciels. De nombreux outils (model-checkers) ont été développés dans ce but: SPIN[2], UPPAAL[3], TINA[4], CADP[5], etc.

[2]http://spinroot.com
[3]http://www.uppaal.com/
[4]http://www.laas.fr/tina
[5]http://www.inrialpes.fr/vasy/cadp/

Une première difficulté liée à l'utilisation de ces techniques de vérification provient du problème bien identifié de l'explosion combinatoire du nombre de comportements des modèles, induite par la complexité interne du logiciel qui doit être vérifié. Cela est particulièrement vrai dans le cas des systèmes embarqués temps réel, qui interagissent avec des environnements impliquant un grand nombre d'entités. Nous n'abordons pas cet aspect dans cet article.

Une autre difficulté est liée à l'expression formelle des propriétés nécessaire à leur vérification. Traditionnellement, cette expression s'effectue à l'aide de formalismes de type logique temporelle comme LTL [27] ou CTL [5]. Bien que ces langages aient une grande expressivité, ils ne sont pas faciles à utiliser par des ingénieurs lors de leurs projets.

Dans l'approche décrite dans cet article, nous proposons de spécifier les propriétés par des observateurs via des patrons de définition de propriétés adapter aux exigences que l'on souhaite vérifier sur les architectures de calcul ciblées. Comme toutes techniques impliquant les méthodes formelles, notre approche repose sur le langage formel, Fiacre et sur la chaîne de transformation, décrite en section 5, des modèles Ruby. Elle repose également sur l'explorateur de modèle Fiacre, OBP Explorer qui exécutent toutes les comportements possibles du modèle. Le langage CDL (Context Description Language) nous permet de décrire les propriétés à vérifier pendant l'exploration du modèle Fiacre.

Dans notre expérimentation, Le modèle Ruby de l'étude de cas est converti en Fiacre. Puis différents types d'exigences, exprimées dans le langage CDL, sont vérifiées.

6.2 Le langage Fiacre

Le langage Fiacre (Format Intermédiaire pour les Architectures de Composants Répartis Embarqués) [14] a été développé dans le cadre du projet TOPCASED[6] comme langage pivot entre les formalismes de haut niveau comme UML, AADL, SDL et les outils d'analyse formelle. L'utilisation d'un langage formel intermédiaire apporte l'avantage de réduire le gap sémantique entre les formalismes de haut niveau et les descriptions manipulées en interne par les outils de vérification comme les réseaux de Petri, les algèbres de processus ou les automates temporisés. Fiacre peut être considéré comme un langage disposant d'une sémantique formelle et servant de langage d'entrée à différents outils de vérification.

Fiacre intègre les notions de *processus* et de *composants* :

- Les automates (*processus*) sont décrits par un ensemble d'états, une liste de transitions entre ces états avec des constructions classiques (affectations des variables, if-elsif-else, while, compositions séquentielles), des constructions non déterministes et des communications par ports et par variables partagées.
- Les composants (*component*) décrivent les compositions de processus. Un système est construit comme une composition parallèle de composants et/ou processus qui peuvent communiquer entre eux par des ports ou variables partagées. Les priorités et les contraintes de temps sont associées aux opérations de communication.

[6]http://www.topcased.org

Les processus Fiacre peuvent se synchroniser de manière synchrone avec ou sans passage de valeur via les ports. Ils peuvent également s'échanger des données via des files de communications asynchrones implantées par des variables partagées. Pour le cas d'étude, nous utilisons ces deux modes de communication dans le modèle Fiacre. L'expression du temps dans le langage Fiacre est basée sur la sémantique des systèmes de transitions Temporisés (TTS) [20]. Toute transition est associée à des contraintes de temps (temps minimum et maximum). Ces contraintes assurent que les transitions sont tirables dans des intervales de temps définies (ni trop tôt, ni trop tard). Dans cet article, nous n'utiliserons pas les aspects temporisés du langage.

6.3 Code Fiacre généré pour le cas d'étude

Le programme Fiacre généré par la chaîne de transfomation Ruby vers Fiacre correspond à l'implantation des règles de traduction décrites en section 5.

Le modèle Fiacre comprend un composant *Calcul* (Listing 1) qui reproupe les instances des processus qui s'exécutent en parallèle (opérateur ||) : `PFor`, `PWhile` et `PUntil`. Nous ajoutons dans le composant une instance du processus `Clock` pour cadencer le comportement des trois processus précédents. Pour cela, le proccessus `Clock` se synchronise avec les processus par le port *Clk*. Ces instances communiquent également par les variables partagées de type `t_fifo` pour les files de messages. Ainsi, les instances des processus `PFor` et `PWhile` partagent la variable $fifoF2W$ déclarée comme une file de message. Les instances de `PWhile` et `PUntil` partagent la file de message $fifoW2U$.

```
component Calcul is
  var
     fifoF2W  : t_fifo,
     fifoW2U  : t_fifo
  port
     Clk :  none
  init
     fifoF2W := {||};
     fifoW2U := {||}
  par
        Clk -> PFor   [Clk] (&fifoF2W)
     || Clk -> PWhile [Clk] (&fifoF2W, &fifoW2U)
     || Clk -> PUntil [Clk] (&fifoW2U)
     || Clk -> Clock  [Clk]
end  Calcul
```

Listing 1: Déclaration Fiacre du composant *Calcul*.

Le comportement de chaque processus est modélisé par un automate. Les états et transitions de la figure 6 sont traduits par des états et transitions Fiacre. Le listing 2 illustre la programmation du processus (`PFor`).

```
process PFor [ck : in none] (&output : t_fifo)
is
states Start, ecrire_data, Vidage, End
var
  noVal  : int, data : t_data
init
```

```
  noVal  := 1;
  data := {index = 0, value = 0};
  to Start

/*--------- ProcFor --------------*/
from Start
   ck;   /* synchro Clk */
   to ecrire_data
from ecrire_data
   //---- ecriture de data si fifo non pleine ---
   if (not (full output)) then
      data.index := noVal;
       data.value := noVal * noVal;
       output := enqueue (output, data);
       noVal := noVal + 1;
       if noVal > NVAL_MAX  then to End end
   end;
     to Start
from End
   ck;  // pour permettre la fin des calculs
     loop
```

Listing 2: Déclaration Fiacre du processus $PFor$.

Pour les communications asynchrones, les opérations $first$, $dequeue$ et $enqueue$ permettent respectivement de lire un message en tête d'une file, de suprimer un message en tête de file et d'écrire un message en queue de file. Chaque processus se voit attribuer une file d'entrée unique. L'écriture d'un message dans cette file ($enqueue$) correspond à un envoi.

Notre schéma de traduction du modèle VHDL à Fiacre nous a contraint à certaines adaptations sémantiques, sans toutefois impacter sur le type de propriétés que nous cherchons à vérifier : du fait qu'il n'est pas possible, dans un même état, en Fiacre, d'accéder à la fois à un port synchrone (celui de l'horloge dans notre cas) et à une variable partagée (FIFO dans notre cas) en écriture, nous avons dû introduire des états non synchronisés sur l'horloge. Cela a pour conséquence une augmentation de l'espace des états, par un entrelacement lié aux libertés d'asynchronismes présentes dans ce modèle. Le comportement réel du circuit est plus contraint et est donc inclu dans cet espace d'exploration. Toutes les propriétés vérifiées sont des propriétés de sûreté, ce qui nous garantit la pertinence des résultats de preuves formulées (ceci n'aurait pas été vrai dans le cas de propriétés d'atteignabilité etc).

Pour cette traduction, le modèle est interprété avec la sémantique suivante : l'ordonnancement des processus concurrents (i.e., s'exécutant en parallèle) repose sur l'entrelacement atomique et non déterministe des transitions de chaque processus (atomique signifiant qu'une transition ne peut pas être suspendue au milieu de son exécution). C'est-à-dire, pour tous les processus possèdant des transitions tirables, une transition est susceptible d'être choisie de manière indéterministe et exécutée de manière atomique. Ensuite une autre transition, parmi l'ensemble des transitions tirables, est à nouveau choisie.

6.4 Exigences à vérifier

Dans notre expérimentation, nous nous intéressons ici à plusieurs exigences que nous souhaitons vérifier sur le modèle Fiacre. Ces exigences traitent de la correction des valeurs calculées, du respect du remplissage des fifos.

La première exigence (*Invariant 1*) concerne la correction des valeurs. Nous vérifions que toutes valeurs lus par le processus (`PUntil`) sont les bonnes valeurs. Pour cela, à toutes les valeurs *data* générées et transmises par (`PFor`) et (`PWhile`), nous leur associons le numéro (*index*) de la valeur. L'invariant 1 s'exprime donc ainsi en CDL :

```
Invariant 1 :
   {PUntil}1:data.index  = {PUntil}1:data.value }
```

Les exigences suivantes expriment que les fifo $fifoF2W$ et $fifoW2U$ ne débordent pas. Elles s'expriment ainsi :

```
Invariant 2 :
     {{Comp}1:fifoF2W.length < FIFO_SIZE and
      {Comp}1:fifoW2U.length < FIFO_SIZE }
```

length permettant de connaître le nombre d'éléments dans une fifo et $FIFO\_SIZE$ étant une constante du programme Fiacre.

Une dernière exigence exprime que le processus (`PUntil`) reçoit la dernière valeru avant de terminer. Elle s'exprime à l'aide d'un patron de définition de propriété. Cette propriétée est transformée en un automate observateur qui est activé durant l'exploration du modèle.

```
property  pte_ProcUntil_lastData  is  {
     evt_PUntil_End   leads_to
     evt_PUntil_receive_lastData
}
```

Cet observateur référence les évènements *evt_PUntil_End* et *evt_PUntil_receive_lastData* déclarés de la façon suivante :

```
predicate PUntil_End is
     { {PUntil}1@End }
event  evt_PUntil_End is
     { PUntil_End  becomes true }
predicate PUntil_receive_lastData is
     { {PUntil}1:data.value = NVAL_MAX }
event  evt_PUntil_receive_lastData is
     { PUntil_receive_lastData  becomes true }
```

$NVAL\_MAX$ étant la dernière valeur calcul par (`PWhile`). Les évènements sont déclarés avec les prédicats *PUntil_End* et *PUntil_receive_lastData*.

Cette propriété est traduite en un automate observateur contenant un état de *rejet* comme illustré figure 7.

Un automate observateur est un automate qui observe l'ensemble des évènements survenant lors de l'exploration du modèle : envois et réception de messages, changement d'états des processus et des valeurs de variables. A chaque exécution d'une transition du modèle ou de l'environnement, l'observateur exécute une transition (si elle existe) correspondante à l'évènement survenu. Si l'observateur possède un état d'erreur *reject*, l'accès à cet l'état signifie que la propriété encodée par l'automate est falsifiée. Une analyse d'accessibilité consiste alors en la recherche de scénarios observés lors de l'exploration du modèle du système conduisant

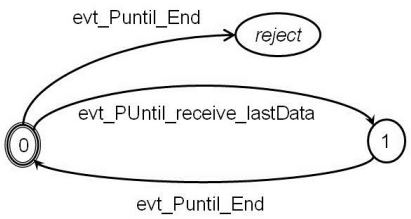

Figure 7: L'observateur correspondant à la propriété *pte_ProcUntil_lastData*.

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 des propriétés plus difficilement exprimables par des logiques temporelles tel que LTL ou CTL.

7. OUTILLAGE DE VÉRIFICATION ET RÉSULTATS

7.1 L'outil OBP

Pour mener les expérimentations, nous mettons en œuvre l'outil OBP Explorer (figure 5). OBP génère un automate observateur pour chaque propriété basé sur les patrons de définition de propriété. Lors de l'exploration du modèle Fiacre, une analyse d'accessibilité est menée sur le résultat de la composition entre l'ensemble des observateurs générés et le modèle. Les invariants sont également vérifiés lors de cette exploration.

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

7.2 Expérimentation

Nous donnons en table 1 les résultats des vérifications menées en fonction du nombre de valeurs traitées. Ce tableau mentionne les temps d'exploration et la complexité en nombre cumulé de configurations et de transitions des systèmes de transitions (SdT) générés durant l'exploration. Ce SdT correspond à l'espace des états nécessaires pour le calcul.

Le langage CDL et l'explorateur OBP permet de faire des mesures sur la performances du circuit de calcul. Nous avons instrumentés le code Fiacre généré avec des variables auxillaires qui donne le nombre de période d'attente pour les procesus PFor et PWhile. Le tableau 2 mentionne, en guise d'exemple, ces temps d'attentes en fonction des taille des fifo pour un nombre de valeurs traités $NVAL\_MAX$ égal à 100.

Nbre de valeurs traitées	Nombre de ticks	Nombre de configurations explorées	Nombre de transitions explorées
1	4	45	52
30	555	4 083	5138
100	5350	42 023	52 948
200	20 700	164 223	206 248
300	46 050	366 423	459 548

Table 1: Complexité du calcul.

Taille des fifo	Nbre attentes PFor	Nbre attentes PUntil
1	5046	5250
2	4945	5250
3	4845	5250

Table 2: Nombre de périodes d'attente pour PFor et PUntil.

Le pouvoir d'expression de CDL permet donc d'observer avec un grain fin, durant les explorations, le comportement du programme (changement des valeurs de variables, des états des processus, contenu des fifos . . .).

8. DISCUSSION ET PERSPECTIVES

Notre étude constitue une première expérimentation qui vise à associer une modélisation de haut niveau, résolument orientée utilisateur (Ruby) à des techniques de vérification formelles, exhaustives. L'enchaînement des phases de modélisation et de vérification s'est révélée opérationnelle. Ces techniques nous ont permis d'aborder à la fois la vérification de fonctionnalités pures, algorithmiques, mais également d'interroger le système sur ses performances. Le concepteur –qui pouvait se reposer "passivement" sur une simulation globale – est directement impliqué ici dans l'expression des besoins et caractérisiques exactes du système, ce qui est déjà un apport méthodologique important. Notre étude nous a également conduit à décrire des exigences qui ont tout lieu de se révéler génériques et applicables sur d'autres systèmes modélisés. En particulier, l'exploration des paramètres comme la taille des FIFOs est une piste intéressante.

Jusqu'ici il est par contre raisonnablement difficile de se prononcer sur la scalabilité de l'approche : des expériences à venir pourrons nous renseigner. Nous savons d'ores-et-déjà que nous devrons recourir à d'autres techniques de vérification formelle dans les cas où l'arithmétique de traitement dépasse les capacités du model-checker utilisé. Dans tous les cas, la synergie des deux techniques de modélisation agiles et de vérification permet d'envisager des expériences rapides, très bénéfiques au développement du model-checker lui-même.

9. CONCLUSION

Nous avons présenté dans cet article une expérience de modélisation conjointe logicielle-matérielle, assistée par model-checking, sur un cas simple. Notre modélisation a été rendue possible par la chaîne de modélisation et de compilation de Modaë Technologies. Nous avons pu formaliser des pro-

priétés attendues par le concepteur, grâce au langage CDL. La vérification formelle de ces propriétés, assurée par OBP, s'est effectuée à la fois sur des propriétés fonctionnelles, mais également sur des propriétés de performances du système. Ces travaux démontrent une utilisabilité certaine des techniques formelles de manière effective, dans une chaîne qui en était dépourvue jusqu'ici.

10. REFERENCES

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[2] G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems.

[3] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[4] A. Butterfield. A denotational semantics for handel-c. *Formal Asp. Comput.*, 23(2):153–170, 2011.

[5] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[6] P. Coussy and A. Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer, 2008. 300p.

[7] A. Davare, H. Hsieh, A. Sangiovanni-Vincentelli, and Y. Watanabe. Simulation based deadlock analysis for system level designs. *Proceedings 42nd Design Automation Conference 2005*, page 260, 2005.

[8] P. Dhaussy, F. Boniol, and J.-C. Roger. Reducing state explosion with context modeling for model-checking. In *13th IEEE International High Assurance Systems Engineering Symposium (Hase'11)*, Boca Raton, USA, 2011.

[9] P. Dhaussy and J.-C. Roger. Cdl (context description language) : Syntaxe et sémantique. Technical report, ENSTA-Bretagne, 2011.

[10] S. A. Edwards. Compiling concurrent languages for sequential processors. *ACM TRANSACTIONS ON DESIGN AUTOMATION OF ELECTRONIC SYSTEMS*, 8:141–187, 2001.

[11] J. Eker and J. Janneck. Caltrop—language report (draft). Technical memorandum, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, 2002. http://www.gigascale.org/caltrop.

[12] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[13] J. Falk, C. Zebelein, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya. Analysis of systemc actor networks for efficient synthesis. *ACM Trans. Embedded Comput. Syst.*, 10(2):18, 2010.

[14] P. Farail, P. Gaufillet, F. Peres, J.-P. Bodeveix, M. Filali, B. Berthomieu, S. Rodrigo, F. Vernadat, H. Garavel, and F. Lang. FIACRE: an intermediate language for model verification in the TOPCASED environment. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008*. SEE, janvier 2008.

[15] R. Ginosar. Fourteen ways to fool your synchronizer. In *ASYNC*, pages 89–97. IEEE Computer Society, 2003.

[16] Grant, Bailey, and Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, USA, 2007. 488p.

[17] D. Große, H. M. Le, and R. Drechsler. Proving transaction and system-level properties of untimed systemc tlm designs. In *MEMOCODE*, pages 113–122, 2010.

[18] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark : A high-level synthesis framework for applying parallelizing compiler transformations. *International Conference on VLSI Design*, 0:461, 2003.

[19] C. Hardebolle and F. Boulanger. Modhelx : A component-oriented approach to multi-formalism modeling. *Science*, pages 1–10, 2007.

[20] T. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *Proceedings of the 1991 REX Workshop*, 1991.

[21] P. Herber, J. Fellmuth, and S. Glesner. Model checking SystemC designs using timed automata. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 131–136, New York, NY, USA, 2008. ACM.

[22] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[23] ITRS. International technology roadmap for semiconductors, design.technical report. 2005.

[24] J. Kunkel. Cossap: A stream driven simulator. In *IEEE International Workshop on Microelectronics in Communications, Interlaken, Switzerland*. IEEE, mar 1991.

[25] E. A. Lee. Disciplined heterogeneous modeling - invited paper. In *MoDELS (2)*, pages 273–287, 2010.

[26] M. Mintz and R. Ekendahl. *Hardware Verification with SystemVerilog?: An Object-oriented Framework*. Springer, 2007. 299p.

[27] A. Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[28] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[29] A. L. Sangiovanni-Vincentelli. The tides of eda. *IEEE Design & Test of Computers*, 20(6):59–75, 2003.

[30] M. Y. Vardi. Formal techniques for systemc verification. *Design*, pages 188–192, 2007.

[31] S. Vercauteren, D. Verkest, G. G. de Jong, and B. Lin. Derivation of formal representations from process-based specification and implementation models. In *ISSS*, pages 16–, 1997.