Mise en œuvre d'un simulateur DEVS utilisant des réseaux de Petri temporisés sur FPGA

Clément Foucher, Vincent Albert

12ème Conférence Internationale de MOdélisation, Optimisation et SIMulation - MOSIM'18 - 27 au 29 Juin 2018 - Toulouse -France «Essor des systèmes connectés»

MISE EN ŒUVRE D'UN SIMULATEUR DEVS UTILISANT DES RÉSEAUX DE PETRI TEMPORISÉS SUR FPGA

Clément Foucher, Vincent Albert

LAAS–CNRS, Université de Toulouse, CNRS, UPS, Toulouse, France
clement.foucher@laas.fr, vincent.albert@laas.fr

RÉSUMÉ : *Une étape du Rapid Control Prototyping peut consister à exécuter une version simulée d'un système de commande en lien avec un environnement réel. Dans ce cadre, le simulateur doit assurer une exécution en temps réel de la simulation de la commande afin de permettre un fonctionnement opérationnel. Cet article propose une mise en œuvre matérielle d'un simulateur à événements discrets qui, si elle ne garantit pas encore le temps réel, permet d'accélérer fortement la simulation pour tendre vers cet objectif. Il s'agit d'un travail en cours s'inscrivant dans notre volonté d'assurer un flot complet de génération d'un système depuis des modèles dans une approche d'ingénierie dirigée par les modèles.*

MOTS-CLÉS : *Simulation à événements discrets, Réseaux de Petri, Accélération matérielle, Rapid Control Prototyping, Temps réel, FPGA*

1 INTRODUCTION

Les noyaux de simulation peuvent être découpés en deux grandes catégories : les simulateurs à temps discret, et les simulateurs à événements discrets. Dans une simulation à temps discret, le temps évolue par pas fixes prédéterminés. Dans une simulation à événements discrets, l'évolution du temps simulé est variable, déterminée par le temps auquel les événements de simulation se produisent.

La simulation à événement discrets peut comporter des avantages en termes de précision et/ou de durée d'exécution de la simulation, selon les cas considérés. Ainsi, dans un modèle dans lequel les variables évoluent lentement, il est possible d'obtenir des pas de simulation longs, et donc moins de pas et ainsi une durée d'exécution réelle plus courte pour arriver au même temps simulé. À l'inverse, si les valeurs des variables se mettent à changer plus rapidement, les événements arriveront plus souvent, conduisant à une précision de simulation plus fine.

Néanmoins, certaines précautions sont à prendre, comme le fait que des événements peuvent se produire en temps nul, conduisant dans le pire des cas à une boucle en temps zéro, et donc à l'impossibilité d'évoluer au delà d'un certain temps simulé. Par ailleurs, la variabilité des pas de temps conduit à une difficulté à prédire la durée réelle de la simulation, celle-ci pouvant être très différente en fonction des vitesses d'évolution des variables au sein du modèle.

Nous étudions la simulation à temps discret au travers d'un environnement de simulation que nous avons conçu : Project DEVS (Albert et Foucher, 2018). Project DEVS est basé sur le formalisme DEVS (*Discrete Event System Specification*) (Zeigler et al., 2000), qui permet de définir de manière mathématiquement formelle des modèles et leur sémantique d'exécution. Notre simulateur est constitué d'un cœur de simulation, historiquement développé en Java, qui génère des classes représentant les modèles, puis les compile et les exécute pour simuler ces modèles. Project DEVS dispose d'une interface utilisateur (*Graphical User Interface* – GUI), également en Java, que nous nommons ProDEVS.

Nous adjoignons actuellement à cet outil la possibilité de distribuer la simulation sur différentes plateformes d'exécution, au travers d'une génération de code en langage C, qui permet une exécution logicielle plus efficace qu'en Java, mais également en langage VHDL, qui permet la génération de circuits logiques dédiés, déployables notamment sur FPGA. La génération matérielle poursuit deux buts. Le premier objectif est un but d'optimisation de la durée d'exécution. En effet, pour les modèles disposant d'un parallélisme inhérent, une exécution matérielle permet de tirer pleinement parti de ce parallélisme. Le second objectif est plus prospectif : il s'agit de faciliter le développement du contrôle d'un système dans un objectif de *Rapid Control Prototyping* (RCP) consistant à générer le code destiné au produit fini directement depuis les modèles de haut niveau.

Pour cela, une étape intéressante consiste à coupler un contrôle simulé à un environnement réel, afin de

vérifier dès la création des modèles de haut niveau que leur comportement est correct. Dans ce cas, il est nécessaire d'avoir une exécution de la simulation du contrôle qui répond à des temps de simulation bornés afin de permettre une exécution en temps réel. dans ce contexte, améliorer la durée d'exécution d'une simulation permet de faciliter l'intégration du modèle simulé dans un environnement réel.

Dans cet article, nous détaillons la mise en œuvre du simulateur matériel dans Project DEVS, et les premiers résultats que nous avons obtenus. Il s'agit d'un travail en cours, certaines parties de la génération de code n'étant pas finalisées, mais les premiers résultats se révèlent encourageant.

Dans la section 2, nous détaillons les motivations nous ayant poussés à développer une mise en œuvre matérielle. La section 3 passe en revue différents travaux déjà réalisés sur la mise en œuvre matérielle de Réseaux de Petri, que nous utilisons comme modèles intermédiaires. La section 4 présente la mise en œuvre que nous proposons, tandis que la section 5 détaille les résultats obtenus. Enfin, dans la section 6, nous concluons sur ce développement tout en indiquant les pistes actuellement suivies par nos travaux sur ce sujet.

2 MOTIVATIONS

L'approche de l'ingénierie dirigée par les modèles (*Model-Based System Engineering* – MBSE) consiste à développer d'abord des modèles de haut niveau, puis à raffiner ceux-ci jusqu'à des modèles plus précis, pour terminer par le système final. La traçabilité entre tous les niveaux est une contrainte majeure qui permet de s'assurer que les modifications effectuées au plus haut niveau d'abstraction se répercutent sur les modèles de plus bas niveau. Dans le cadre de l'ingénierie système, les exigences et les spécifications formant le cahier des charges peuvent même parfois être considérées comme le modèle de plus haut niveau.

Une approche de prototypage virtuel (*Virtual Prototyping* – VP) consiste à modéliser la totalité du produit fini, incluant le contrôle et l'éventuelle partie physique du dispositif, ainsi que son environnement. Pour le système de contrôle, la génération automatisée constitue l'approche dite de *Rapid Control Prototyping* et permet de passer directement des modèles au code exécutable. Dans ce contexte, la co-simulation entre les modèles du contrôle et les modèles de son environnement peut évoluer au cours du cycle de production pour que les modèle du contrôle puissent communiquer avec un environnement physique réel.

Dans ce cadre, il est nécessaire que le simulateur qui porte les modèles du contrôle soit à même d'assurer que l'évolution du temps simulé suive le temps réel. Pour cela, il est nécessaire de borner l'exécution de la simulation, et de s'assurer que la puissance de calcul soit suffisante pour assurer ces bornes. Or, Project DEVS a été historiquement développé en Java, alors que la gestion du temps réel n'était pas envisagée dans l'outil. Le niveau d'abstraction de Java rend particulièrement difficile, voir impossible en raison des processus non maîtrisés tels le *garbage collector*, la garantie d'un temps réel.

L'objectif de notre mise en œuvre matérielle est d'exploiter au maximum les capacités de parallélisme et d'optimisation propres aux systèmes numériques matériels pour accélérer la simulation. Par système numérique matériel, on entend un circuit logique dédié à une application, permettant de prendre en compte ses spécificités, par opposition à un traitement logiciel effectué sur un processeur séquentiel générique. Les circuits numériques matériels permettent deux améliorations principales par rapport à un traitement logiciel.

Tout d'abord, le parallélisme. En effet, dans un circuit logique, des traitement indépendants peuvent être effectués de manière simultanée à chaque cycle d'horloge, là où un processeur ne permet l'exécution que d'un seul calcul à la fois. Dans des systèmes multi-cœurs et/ou multiprocesseurs, certaines opérations peuvent évidemment être parallélisées, mais la granularité reste au niveau du processus ou du thread, là où le matériel peut descendre au niveau du calcul élémentaire. De plus, même dans les systèmes disposant de plusieurs unités de calcul, celles-ci restent en nombre limité, à l'exception notable des architectures many-cores et des GPU.

Par ailleurs, le second apport principal de l'utilisation d'une architecture logique dédiée consiste en la personnalisation à l'envie de l'architecture en elle-même. En effet, un processeur est limité par sa généricité, les cycles d'exécution consistant à aller chercher l'instruction à utiliser et les données (*fetch*), effectuer le calcul (*execute*), puis stocker le résultat (*store*), souvent en passant par un *pipeline* comportant de nombreux niveaux. En revanche, une circuit dédié ne dispose pas de cette limitation, le circuit étant directement conçu pour réaliser l'opération en question.

Les circuits dédiés peuvent être réalisés sous une forme définitive lorsque ceux-ci ont vocation à être utilisés à grande échelle. On parle alors d'ASIC (*Application-Specific Integrated Circuit*). Ils peuvent également être réalisés sous forme reprogrammable sur des composants de type FPGA (*Field-Programmable Gate Array*). Les FPGA offrent l'avantage d'un cycle très court entre la conception du circuit et sa disponibilité, pouvant aller de quelques minutes à quelques heures selon la complexité du

circuit à synthétiser. Un FPGA restant un circuit générique, il offre des performances moindres qu'un ASIC, mais qui restent tout de même extrêmement compétitives par rapport à un processeur, notamment pour des applications utilisant massivement le parallélisme.

Notre objectif dans cet article est de proposer une mise en œuvre de couples modèles/simulateur pour des modèle au formalisme DEVS. Ainsi, pour un modèle saisi dans l'interface ProDEVS, il s'agit de générer automatiquement un composant matériel comprenant le modèle associé à un simulateur afin d'être utilisable de manière autonome pour réaliser la simulation du modèle au sein de l'environnement Project DEVS.

3 TRAVAUX ANTÉRIEURS

Notre approche de génération des modèles de simulation se base sur des modèles intermédiaires en Réseaux de Petri temporisés. En effet, nous utilisons les Réseaux de Petri (RdP) pour permettre une vérification formelle des modèles DEVS, notamment afin de vérifier que ceux-ci ne disposent pas de boucle infinie en temps zéro. Pour cela, nous générons automatiquement des modèles en RdP représentant les modèles DEVS saisis dans l'interface ProDEVS. La vérification de la correspondance entre les modèles DEVS et leur équivalent en RdP est réalisée par bisimulation. Nous avons précédemment présenté ces travaux dans (Albert et Foucher, 2017). Ce sont donc ces modèles intermédiaires que nous utilisons comme base pour la génération matérielle. L'étude des travaux antérieurs que nous avons réalisée balaie donc le domaine de la génération matérielle depuis les Réseaux de Petri.

Dans (Węgrzyn et Węgrzyn, 2011), deux approches pour la mise en œuvre de RdP en VHDL sont proposées. La première méthode utilise un *process* général, synchrone, comprenant la totalité de la description du RdP. Celui-ci est découpé en trois blocs : le premier permet la définition des transitions à tirer, le second met à jour le marquage des places et le troisième met à jour les sorties. Ces blocs communiquent entre eux par des variables. Bien que cela produise un code VHDL correct et synthétisable, cette méthode produit un seul fichier dont la taille croit avec la taille du RdP. Par ailleurs, le détail de la représentation en places et transitions n'apparaît pas explicitement, celle-ci étant noyée dans un flot de variables.

La seconde méthode proposée représente les places et les transitions sous la forme de composants, les liant par des signaux représentant des arcs. Cette méthode préserve la visibilité de la représentation places/transitions dans le code final, produisant une vue du modèle proche de la représentation du RdP. De plus, cette méthode est plus proche de la représentation basée composants que l'on retrouve fréquemment dans les circuits numériques. Il semble s'agir de la méthode généralement utilisée dans la littérature, comme suggéré dans (Gomes et al., 2007). C'est donc cette seconde méthode que nous utiliserons pour la représentation.

Dans (Soto et Pereira, 2005), les auteurs présentent une méthode pour mettre en œuvre les RdP dans un FPGA. Leur description met l'accent sur la réduction de la consommation de ressources matérielles. Ainsi, il s'agit de minimiser le nombre de CLB (*Configurable Logic Bloc*) utilisés en mutualisant ceux-ci pour les places du RdP. Néanmoins, cela apporte une complexité intrinsèque au code généré, et notre approche n'est pas pour l'instant axée vers l'optimisation des ressources. Ainsi, si cet article apporte des points de vue intéressants, nous n'irons pas jusqu'à mettre en œuvre cette optimisation.

Dans l'article (Bukowiec, 2013), les auteurs proposent une manière de mettre en œuvre un RdP en utilisant une approche GALS (*Globally Asynchronous-Locally Synchronous* – Globalement asynchrone, localement synchrone). Ceci permet d'avoir différentes parties d'un RdP utilisant chacune une horloge indépendante des autres. La synchronisation est effectuée entre les domaines d'horloge en utilisant des buffers. Cet article porte un regard très intéressant sur le découpage en sous-réseaux et l'indépendance des sous-réseaux entre eux.

4 LA PLATEFORME MATÉRIELLE

Pour notre mise en œuvre, nous avons besoin d'un circuit programmable pour la réalisation du couple modèle/simulateur, mais également d'une passerelle permettant la communication du système avec l'environnement afin de pouvoir commander celle-ci à distance directement depuis l'interface ProDEVS. Pour cette partie, dont les performances ne sont pas aussi cruciales que l'exécution du modèle en lui-même, nous avons choisi d'utiliser une mise en œuvre logicielle afin de disposer des outils existant pour la gestion du réseau, notamment la pile TCP/IP qui permet de disposer immédiatement de communications standardisées par le biais de *sockets*.

Pour cela, nous avions besoin d'une plateforme permettant aisément de mélanger le matériel au logiciel, et facilitant l'accès à un réseau. Nous avons donc choisi la plateforme de développement ZedBoard (Avnet, 2018). En effet, celle-ci comporte un système sur puce (*System on Chip* – SoC) Zynq (Xilinx, 2017), qui comprend un système matériel reprogrammable (FPGA) et un processeur situés sur un même circuit. Elle dispose également de toute la connectivité nécessaire à un branchement

réseau, notamment une interface Ethernet.

Ainsi, comme présenté sur la Figure 1, notre mise en œuvre est constituée de deux parties, une logicielle s'exécutant sur un processeur et une matérielle fonctionnant sur le FPGA, les deux communiquant via un bus AXI.

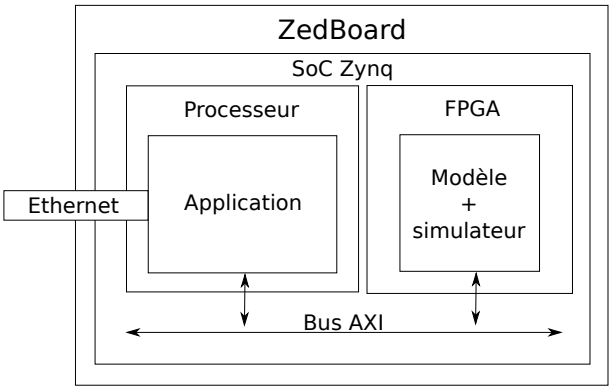

Figure 1 – Architecture de la plateforme matérielle.

4.1 Le bloc logiciel

Le bloc logiciel fonctionne sur un système d'exploitation FreeRTOS (Real Time Engineers Ltd., 2018), système ouvert permettant le déploiement de logiciels multi-threadés, accompagné de la bibliothèque LwIP (Dunkels, 2018), qui fournit une pile réseau TCP/IP. Le système logiciel que nous avons développé est générique, c'est-à-dire que le cœur est commun à tous les déploiements que nous effectuons depuis Project DEVS. Ce cœur commun comporte la partie communication avec l'interface, reposant sur un protocole développé sur le socle des sockets TCP/IP, et permettant de recevoir des ordres de l'interface, et d'envoyer des résultats de simulation lorsque ceux-ci sont disponibles.

À coté de ce cœur générique, des fichiers de configurations sont générés automatiquement par Project DEVS, qui décrivent les caractéristiques du modèle, sous la forme de fichiers d'en-tête qui sont automatiquement inclus par le cœur qui les utilise pour se configurer. Ces fichiers décrivent l'interface du modèle simulé, c'est-à-dire ses sorties, et les registres associés à chacune d'entre elle afin que le cœur soit à même de monitorer leur état, et ainsi de transmettre les valeurs générées sur les sorties à l'interface ProDEVS.

Le cœur dispose de différents threads. Parmi ceux-ci, on note principalement un serveur TCP/IP qui attend les commandes de l'interface, un thread de monitoring du circuit matériel vérifiant en permanence si des sorties ont été émises par le simulateur et un thread client TCP/IP qui émet les résultats en direction de l'interface. Tous ces threads communiquent entre eux par des boîtes aux lettres, ce qui permet un fonctionnement asynchrone, les threads ne se déclenchant que lorsqu'ils ont un travail à effectuer. Ainsi, le thread de monitoring du circuit matériel ne démarre que lorsque la simulation est lancée, et le thread client TCP/IP ne se déclenche que lorsqu'il y a des données à émettre.

4.2 Le bloc matériel

Le bloc matériel constitue l'élément principal de notre mise en œuvre : le simulateur. Comme indiqué précédemment, le bloc contient à la fois le modèle simulé et le simulateur associé, rendant le bloc totalement autonome en termes d'exécution. Comme présenté dans (Albert et Foucher, 2017), nous utilisons les Réseaux de Petri temporisés comme pivot dans nos descriptions de modèles DEVS. Ceci nous permet de garantir que le modèle utilisé pour la validation formelle est le même que celui utilisé dans la simulation.

Le simulateur mis en œuvre est donc un Réseau de Petri temporisé, composé de plusieurs sous-réseaux correspondant aux modèles atomiques DEVS. La hiérarchie des modèles couplés est mise à plat, celle-ci n'étant pas nécessaire à la simulation proprement dite. Ainsi, tous les composants atomiques se retrouvent au même niveau dans le simulateur matériel. En revanche, l'isolation des modèles atomiques en composants est préservée, et ce pour deux raisons. Tout d'abord, cela nous permet d'optimiser légèrement la mise en œuvre, en mutualisant au niveau du composant certains signaux qui peuvent apparaître à de multiples reprises au sein de celui-ci. Ensuite, dans une perspective d'évolution future consistant à permettre la modification des composants de la simulation dynamiquement, et qui nécessitera de conserver cette séparation.

Par ailleurs, aux côtés de ces composants matériels représentant les composants atomiques DEVS, on trouve différents autres composants faisant partie du simulateur. Ainsi du coordinateur, composant supplémentaire destiné à rythmer la simulation en s'assurant que tous les composants ont terminé leur cycle de simulation avant de commencer le suivant, également décrit dans (Albert et Foucher, 2017). En plus de ces éléments, nous ajoutons deux composants principaux : le gestionnaire de temps, et le gestionnaire d'exécution.

Dans notre mise en œuvre, les composants évoluent de manière indépendante les uns des autres au cours d'un cycle de simulation, ceux-ci étant synchronisés entre deux cycles par le coordinateur. Ainsi, plusieurs transitions du RdP appartenant à des composants différents peuvent être tirées au cours d'un cycle d'horloge. Du point de vue du RdP global (comprenant tous les RdP partiels), cela implique

que plusieurs transitions sont tirées simultanément. Cela ne pose pas de problème car, dans notre cas, les transitions appartenant à des composants différents dépendent de ressources (places, variables) indépendantes. Néanmoins, ce comportement ne pourrait pas être généralisé à n'importe quel RdP.

4.2.1 Les composants atomiques

Les composants atomiques mettent en œuvre des Réseaux de Petri temporisés partiels, dans le sens où certains arcs proviennent de l'extérieur du composant, et d'autres en sortent. Un composant atomique est donc constitué de places et de transitions reliés par des arcs. Afin de s'approcher au plus près de la structure des RdP, les places sont représentées par des mémoires contenant l'état de celles-ci, les transitions par des composants asynchrones et les arcs par des équations logiques reliant les places aux transitions. Comme notre modèle de RdP n'utilise que des places binaires, un simple bascule à un seul bit suffit à mémoriser l'état. Les bascules sont synchrones à une horloge, marquées et démarquées par le jeu des équations booléennes reliant celles-ci.

Pour la gestion du temps, notre RdP temporisé utilise des intervalles réduits à un seul point ($[t_a : t_a]$) associés aux transitions. Une transition doit être activée depuis un temps t_a pour être tirable. Le temps pris en compte est bien entendu le temps simulé, géré donc par le simulateur.

Afin de simplifier la coordination entre les composants, nous avons fait le choix d'externaliser la gestion du temps dans un composant spécialisé. En effet, dans un RdP temporisé, le temps avance au gré du tir des simulation temporisées, la transition ayant le plus faible t_a étant tirée lorsque aucune transition non temporisée n'est sensibilisée. Ainsi, dans notre mise en œuvre, les transitions temporisées fournissent leur t_a initial et leur état (sensibilisée ou non), et attendent une instruction extérieure leur indiquant qu'elles peuvent être tirées. Le choix du tir n'est donc pas fait dans le composant.

Chaque composant atomique comportant des transitions temporisées dispose donc de sorties indiquant si une transition temporisée est sensibilisée, et le t_a associé le cas échéant. On notera que, par construction, au maximum une transition temporisée peut être sensibilisée à la fois au sein d'un composant atomique. Il n'est donc nécessaire de prévoir qu'un seul port de ce type, même si plusieurs transitions temporisées existent au sein du composant.

4.2.2 Le gestionnaire de temps

Le gestionnaire de temps est en charge de deux éléments majeurs de la simulation : l'avance du temps de simulation, et le tir des transitions temporisées. Ce composant dispose d'autant d'entrées qu'il y a de composants comportant des transitions temporisées. À chaque pas de simulation, le gestionnaire de temps vérifie si aucune transition non temporisée n'est sensibilisée. Si c'est le cas, celui-ci démarre, et calcule le t_a minimal parmi toutes les transitions temporisées sensibilisées. On notera que, pendant cette période, le RdP est figé, car il n'y a aucune transition non temporisée sensibilisée, et toutes les transitions temporisées sont en attente d'une autorisation du gestionnaire de temps pour être tirées.

Lorsque le t_a minimal $t_{a_{min}}$ a été déterminé, le gestionnaire de temps incrémente le temps de simulation de ce même $t_{a_{min}}$, diminuant du même coup tous les t_a de toutes les transitions temporisées sensibilisées de cette valeur. Ensuite, il autorise la ou les transitions pour lesquelles $t_a = t_{a_{min}}$ à être tirées, débloquant du même coup le RdP qui peut alors continuer la simulation.

4.2.3 Le gestionnaire d'exécution

Le gestionnaire d'exécution est a pour but de gérer l'avancement de la simulation : démarrage et arrêt de la simulation. Afin de permettre de gérer la simulation, celui-ci délivre une autorisation d'exécution qui est utilisée par le réseau places-transitions pour inhiber ou libérer son évolution. C'est lui qui réagit aux ordres en provenance du bloc logiciel via les registres du composant matériel.

Celui-ci effectue deux tâches principales : réagir aux ordres de démarrage et de suspension de l'exécution du simulateur en provenance de l'interface utilisateur via le réseau et la partie logicielle, et arrêter le simulateur lorsque le temps maximal est atteint. En effet, lorsque l'on génère un composant matériel, la liberté est donnée de définir le nombre de bits des signaux, là où la taille de ceux-ci dans un code logiciel est fixée par l'architecture à un multiple d'octets. Le choix de ce nombre de bits est donc défini en fonction du temps maximal de simulation défini par l'utilisateur. Néanmoins, cela entraînera une limite fixe, physique, au temps qu'il sera possible d'atteindre dans la simulation, et il convient de s'assurer que celle-ci ne sera jamais dépassée au risque d'entraîner des résultats imprédictibles. Le gestionnaire d'exécution se charge donc de bloquer l'exécution du RdP lorsque le temps maximal est atteint afin d'éviter ce comportement.

4.3 Limites de la génération

La génération automatique de code permet de construire automatiquement le réseau places-transitions ainsi que la gestion du temps. Les conditions supplémentaires, ou gardes, du réseau de Petri sont également prises en compte lors de la génération au-

tomatique. En revanche, une autre partie cruciale du code, à savoir les actions, n'est pour l'instant pas générée automatiquement sauf pour certaines actions simples du type incrément d'une variable. Mais des actions plus complexes requièrent pour l'instant que l'utilisateur tape à la main le code correspondant. Cette partie fait actuellement l'objet d'études de notre part, notamment au sujet de l'utilisation d'outils de *High Level Synthesis* (HLS), qui permettent la génération automatisée de code matériel depuis du code logiciel.

5 EXEMPLES DE DÉPLOIEMENT ET RÉSULTATS

Nous avons testé cette génération automatisée sur deux exemples représentant des cas d'utilisation très différents. Dans les deux cas, les actions des modèles restent simples afin de permettre leur génération automatisée.

Le premier exemple est un exemple classique DEVS : le générateur-buffer-processeur (Gen-Buf-Proc), composé de trois modèles très simples. Le générateur émet des événements à intervalle fixe, le buffer stocke le nombre d'événements, et le processeur les consomme à intervalles fixes. Selon les durées définies du générateur et du processeur, le nombre d'événements dans le buffer croit à l'infini ou oscille entre 0 et 1. Dans notre cas, les durées ont été définies respectivement à 2 et à 3 unités de temps.

Un autre modèle plus complexe met en œuvre le jeu de la vie, en 8×8 cellules toriques. Chaque cellule est connectée à ses 8 voisines afin de mettre à jour son état à chaque cycle de simulation. Ce modèle est donc très connexe, car il comporte de très nombreuses connexions entre les composants, et très parallélisable, car toutes les cellules peuvent s'enquérir de l'état de leurs voisines simultanément.

Le déploiement des modèles en VHDL comporte quatre étapes : la génération du code, la synthèse (ou compilation du code), la programmation de la cible et l'exécution à proprement parler. L'utilisation de code logiciel est similaire mais ne comporte que trois étapes, car l'unité d'exécution du modèle est la même que celle qui génère le code : le processeur. On n'a donc pas de phase de programmation de la cible. Il est à noter que les modèles logiciels sont actuellement générés en Java, et pourraient probablement gagner à utiliser un autre langage plus rapide, mais les ordres de grandeur obtenus restent néanmoins pertinents. Les exécutions ont été réalisées sur une durée de 100 000 unités de temps.

Le Tableau 1 présente les résultats obtenus sur le Gen-Buf-Proc, tandis que le Tableau 2 présente ceux obtenus sur le jeu de la vie. La colonne *sw* présente les résultats obtenus sur une exécution logicielle sur un PC équipé d'un processeur à 3.2 GHz. La colonne *hw1* présente les résultats sur une exécution matérielle libre, c'est-à-dire sans récupération des résultats intermédiaires. Enfin, la colonne *hw2* présente les résultats obtenus avec la même exécution matérielle, mais en récupérant toutes les valeurs intermédiaires des variables internes : taux de remplissage du buffer pour le Gen-Buf-Proc, et état de toutes les cellules pour le jeu de la vie. On note ainsi un très fort ralentissement entre les deux exécutions matérielles dues au temps de communication avec le GUI par le réseau.

Néanmoins, quelle que soit l'exécution matérielle, on note la chose suivante : Sur un modèle très simple de type Gen-Buf-Proc, aucun gain n'est réalisé au passage en matériel, en raison du très fort impact de la durée de compilation du code matériel. Par contre, la durée d'exécution à proprement parler est diminuée d'un ordre 100.

Sur un modèle très connexe et fortement parallélisable en revanche, on note un gain en termes de durée d'exécution d'un facteur 10 lors du passage en matériel. Plus encore, on remarque que les durées totales du matériel sont constituées presque exclusivement du temps de compilation du code, temps qui ne change pas lorsque l'on augmente les durées d'exécution. Un accroissement de la durée de simulation augmenterait donc encore le gain du passage en matériel.

6 CONCLUSION ET PERSPECTIVES

En conclusion, on peut dire que selon le type de modèle que l'on simule, un passage à une exécution matérielle peut être envisagée. Les critères qui justifient ce déploiement sur FPGA sont : une grande connexité du modèle, une forte faculté à la parallélisation (évolutions simultanées et indépendantes au sein du modèle), et un temps simulé important. L'exécution matérielle est handicapée par une durée de synthèse du code très longue, mais qui ne s'accroît pas avec la durée simulée.

Le travail n'est pas terminé, car une grosse partie du travail reste à faire concernant la génération automatisée des actions associées au modèle DEVS. Pour cela, nous étudions actuellement la génération de code utilisant les outils de HLS.

On note aussi que notre mise en œuvre n'est utilisable que sur les Réseaux de Petri représentant des composants DEVS issus de Project DEVS. Ainsi, un certain nombre de restrictions sont appliquées, telles les places binaires ou les intervalles de temps ponctuels. De la même manière, certaines libertés sont prises avec le modèle de calcul des RdP comme la possibilité de tirer des transitions simultanément dans certains

Étape	Durée (sw)	Durée (hw1)	Durée (hw2)
Génération du code	4.3 ms	4.7 ms	4.8 ms
Compilation du code	844.0 ms	102 s	102 s
Programmation de la cible	N/A	3 s	3 s
Durée de la simulation	624.0 ms	6.0 ms	1.0 s
Total	1.5 s	105.0 s	106.0 s

Tableau 1 – Durées de génération et d'exécution pour le modèle du générateur-buffer-processeur.

Étape	Durée (sw)	Durée (hw1)	Durée (hw2)
Génération du code	85.3 ms	1718.6 ms	1717.5 ms
Compilation du code	2.8 s	655 s	663 s
Programmation de la cible	N/A	3 s	3 s
Durée de la simulation	5194.8 s	113.0 ms	8.58 s
Total	5197.7 s	659.8 s	676.3 s

Tableau 2 – Durées de génération et d'exécution pour le modèle du jeu de la vie.

cas. Tout ceci interdit d'utiliser notre mise en œuvre pour des Réseaux de Petri temporisés génériques, certaines précautions devant être prises.

Un autre travail à réaliser concernera le bornage des durées d'exécution. En effet, dans une perspective d'utilisation des modèles dans un contexte de Rapid Control Prototyping, les modèles doivent être capables de s'exécuter en temps réel. Cette partie reste délicate à réaliser, car nos modèles sont à événements discrets. Ainsi, contrairement à des modèles à temps discret dans lesquels il suffit de borner une itération de la boucle d'exécution pour borner la totalité de la simulation, les modèles à événements discrets utilisent des pas de temps variables. Borner une itération de la boucle d'exécution délimite ainsi un temps réel variable, en fonction de la durée représentée dans cette boucle.

Ce travail reste donc ouvert, et ne représente que la première étape de l'extension de notre simulateur Project DEVS à la simulation matérielle.

RÉFÉRENCES

Albert, V. et Foucher, C. (2017). Formal framework for discrete-event simulation, *The 20th World Congress of the International Federation of Automatic Control*.

Albert, V. et Foucher, C. (2018). Site web prodevs. **URL:** *https://www.laas.fr/projects/prodevs/*

Avnet (2018). Site officiel zedboard. **URL:** *http://www.zedboard.org/*

Bukowiec, A. (2013). Distributed control systems design as petri nets for fpgas, *International Journal of Design, Analysis and Tools for Integrated Circuits and Systems (IJDATICS)* **4**: 1–9.

Dunkels, A. (2018). lwip - a lightweight tcp/ip stack. **URL:** *https://savannah.nongnu.org/projects/lwip/*

Gomes, L., Costa, A., Barros, J. et Lima, P. (2007). From petri net models to vhdl implementation of digital controllers, *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*, pp. 94–99.

Real Time Engineers Ltd. (2018). Site officiel freertos. **URL:** *https://www.freertos.org/*

Soto, E. et Pereira, M. (2005). Implementing a petri net specification in a fpga using vhdl, *Design of Embedded Control Systems*, Springer US, pp. 167–174.

Węgrzyn, M. et Węgrzyn, A. (2011). Penlogic – system for concurrent logic controllers design, *in* M. Adamski, A. Barkalov et M. Węgrzyn (eds), *Design of Digital Systems and Devices*, Vol. 79 of *Lecture Notes in Electrical Engineering*, Springer Berlin Heidelberg, pp. 215–228.

Xilinx (2017). Zynq-7000 all programmable soc data sheet: Overview. **URL:** *https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf*

Zeigler, B. P., Kim, T. G. et Praehofer, H. (2000). *Theory of Modeling and Simulation*, 2nd edn, Academic Press, Inc., Orlando, FL, USA.