# Simulation distribuée hétérogène de réseaux de Petri DEVS

Clément Foucher, Vincent Albert

# Simulation distribuée hétérogène de réseaux de Petri DEVS

# Heterogeneous distributed simulation of DEVS Petri nets

Clément Foucher Vincent Albert
LAAS–CNRS, Université de Toulouse, CNRS, UPS, Toulouse, France
7, avenue du Colonel Roche – BP 54200 – 31031 Toulouse cedex 4, France
{Clement.Foucher ; Vincent.Albert}@laas.fr

**Résumé :**

Nous présentons dans cet article la conception d'une plateforme de simulation distribuée de modèles DEVS (Discrete Event System Specification) basée sur l'exécution de réseaux de Petri temporisés qui communiquent au travers d'une connexion réseau. Dans notre proposition, l'exécution des réseaux de Petri est déployée sur une architecture distribuée composée de nœuds logiciels PC et/ou matériels FPGA (Field-Programmable Gate Array) par une mise en œuvre des réseaux de Petri en C et en VHDL. En outre, ce travail s'inscrit dans la perspective future de pouvoir connecter les modèles avec des équipements réels.

**Mots-clés :**

Simulation distribuée, Cosimulation logicielle/matérielle, Réseaux de Petri.

**Abstract:**

In this article, we present the design of a distributed DEVS (Discrete Event System Specification) model simulation platform based on the execution of timed Petri nets that communicate through a network connection. In our proposal, the execution of the Petri nets is deployed on a distributed architecture made of software PC and/or hardware FPGA (Field-Programmable Gate Array) nodes using a C and a VHDL implementation of Petri nets. In addition, this work is part of the future perspective of being able to connect models with real equipments.

**Keywords:**

Distributed Simulation, Hardware/Software Cosimulation, Petri Nets.

## 1 Introduction

Les plateformes de simulation pour la validation des systèmes cyber-physiques sont communément classifiées selon la nature du système de contrôle, la nature du procédé contrôlé et le support matériel sur lequel le système de contrôle est déployé. Le prototypage rapide de contrôleurs (Rapid Control Prototyping, RCP), permet de vérifier et de valider les réponses du système de contrôle le plus tôt possible, à l'aide de tests effectués dans des conditions réelles d'utilisation. Le système de contrôle est sous la forme d'un modèle abstrait puis mis en œuvre sur un support matériel capable de fournir une réponse en temps réel aux entrées et aux informations fournies par le procédé réel.

Nous avons aujourd'hui pour objectif la conception d'une plateforme permettant de déployer rapidement et de façon modulaire une simulation sur des architectures hétérogènes (matérielle/logicielle, équipement réel/virtuel).

Des plateformes de simulation distribuées DEVS ont déjà été proposées par le passé. La plateforme MECSYCO [4] utilise un environment multi-agents dans lequel DEVS sert de langage pivot pour l'intégration de l'hétérogénéité des composants de la simulation. Jusqu'à présent, cette plateforme ne propose pas de distribuer la simulation sur différentes ressources de calcul hétérogènes. L'hétérogénéité est assurée uniquement au niveau des langages utilisés pour la description de modèles par des techniques de co-simulation logicielle. Le même type de plateforme, supportant cette fois l'exécution distribuée de la simulation architecturée autour d'un bus DEVS, a été proposée par [9]. Plus récemment, [8] propose une plateforme basée sur Eclipse qui utilise des microservices pour l'exécution distribuée sur plusieurs cœurs d'une même machine de modèles DEVS homogènes.

D'une part, aucune de ces approche ne propose une sémantique formelle de l'intégration des composants de simulation. D'autre part, aucune ne prend en compte de ressouce de calcul matérielle en tant que nœud de simulation.

Dans ce contexte, nous souhaitons adopter une approche rigoureuse de décomposition du modèle et des communications entre les composants de la simulation. Par ailleurs, l'un de nos objectifs est la prise en compte de l'hétérogénéité en termes de formalismes, de languages et de ressources de calcul, notamment matérielles.

Précédemment, nous avons proposé dans [3] un codage en réseau de Petri (RdP) temporisé d'un modèle DEVS (Discrete-Event System Specification) et de sa sémantique d'exécution. Parallèlement, nous avons développé une mise en œuvre de réseaux de Petri temporisés en VHDL et une mise en œuvre en C. À ce jour, notre outil ProDEVS transforme automatiquement un modèle DEVS en un réseau de Petri temporisé qui est ensuite déployé soit sur une plateforme logicielle (PC) soit sur une plateforme matérielle (FPGA – Field-Programmable Gate Array). Par exécution du réseau de Petri, les résultats de simulation sont fournis à l'utilisateur. Le principal avantage de cette approche dans laquelle à la fois les modèles simulés et le noyau de simulation (l'ordonnanceur) sont codés par des réseaux de Petri est que nous obtenons une modélisation formelle de la sémantique des modèles construits dans ProDEVS.

Ainsi, notre objectif est de décomposer le réseau de Petri obtenu à ce jour en sous-réseaux de Petri et de distribuer les sous réseaux de Petri sur des plateformes différentes. Pour cela nous avons adopté une approche de composition basée sur la notion de fusion de place qui assure la préservation de la sémantique du réseau de Petri global par la décomposition en sous réseaux de Petri. De nombreuses références peuvent être trouvées sur la composition par fusion de places ou de transitions dans la thèse de Chehaibar [5].

Dans la section suivante, nous présentons les concepts du simulateur distribué. Dans la section 3, nous donnons les éléments de mise en œuvre logicielle et matérielle des réseaux de Petri. Enfin, la section 4 discute des conclusions et des perspectives de ce travail.

# 2 Concepts du simulateur distribué

La composition par fusion de place, également appelée composition asynchrone, a été utilisée pour la réalisation de ce codage. Le principe était que chaque composant atomique DEVS soit transformé en un réseau de Petri avec des places d'interface et un réseau de Petri supplémentaire permettait la synchronisation des composants atomiques selon la sémantique souhaitée (Classic DEVS ou Parallel DEVS). Les réseaux de Petri étaient alors fusionnés au travers de leurs places d'interface et le réseau de Petri global obtenu était mis en œuvre en C ou en VHDL. Le codage complet est rigoureusement détaillé dans [3].

Puis, nous avons dans [1] prouvé formellement l'équivalence entre un modèle DEVS et son codage en réseau de Petri par la technique de bisimulation. Nous avons ainsi montré que le codage que nous avons réalisé préserve les branchements et les états.

Aujourd'hui, nous proposons une mise en œuvre d'un réseau de Petri qui correspond non plus au réseau de Petri global mais à un sous ensemble du modèle DEVS (de un à plusieurs composants atomiques). Un modèle DEVS est alors représenté par un ensemble de mises en œuvre de sous réseaux de Petri qui communiquent au travers d'une architecture distribuée.

## 2.1 Architecture distribuée

Le simulateur distribué consiste en un ensemble de nœuds exécutant chacun un simulateur partiel. Par « simulateur partiel », on entend un simulateur en charge d'un ou plusieurs composants atomiques DEVS, ne constituant pas la totalité des composants participant à la simulation. Comme présenté sur la figure 1, les simulateurs partiels sont composés d'une partie communication et d'une partie gestion du réseau de Petri. Ces deux parties sont asynchrones, par exemple réalisées sous la forme de threads communiquant par des boîtes aux lettres pour une mise en œuvre logicielle.

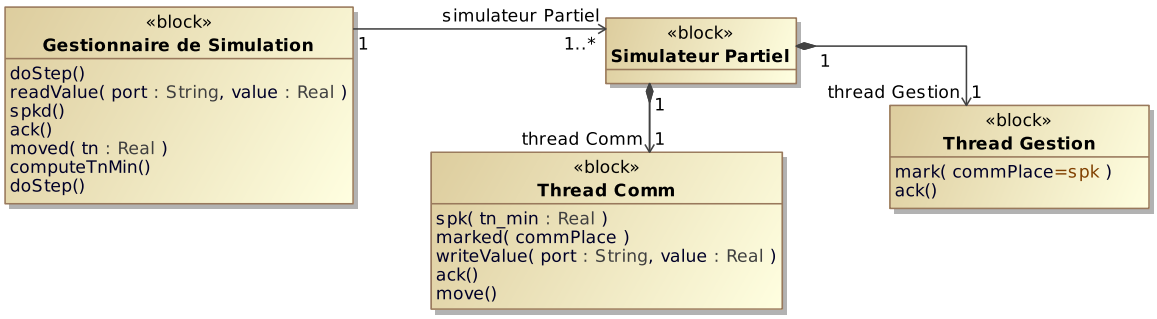



Figure 1 – Architecture du simulateur distribué

Le simulateur distribué est architecturé autour d'un gestionnaire de simulation en charge de la synchronisation des différents simulateurs partiels et de leur communication. La communication entre différents composants DEVS fonctionnant sur le même nœud reste à la charge du simulateur partiel. En revanche, la communication entre composants simulés sur des nœuds différents passe par le gestionnaire de simulation. Dans tous les cas, la synchronisation des composants est coordonnée par le gestionnaire de simulation.

La communication entre les nœuds et le gestionnaire de simulation est effectuée par une connexion de type socket TCP. Nous utilisons un médium de transmission de type Ethernet, mais l'utilisation d'un protocole de communication de haut niveau permet de s'affranchir de la couche physique et autorise l'utilisation d'autres supports de communication si nécessaire.

Afin de correctement coordonner les composants, le gestionnaire de simulation doit avoir accès à l'information indiquant la fin d'un pas de simulation, c'est-à-dire le moment où tous les composants ont traité leurs entrées et mis à jour leur état.

Chaque composant dispose d'une place **_spk* dont le marquage indiquele début d'un cycle de communication, et les composants utilisent leur place **_spkd* pour indiquer qu'ils ont terminé leur cycle de communication. De la même manière, les places **_mv* et **_mvd* délimitent un cycle de mise à jour de l'état du composant.

Le marquage des places permettant de débuter le cycle de communication dans les composants est effectué lors de la réception d'un événement *speak*, et le marquage des places pour le cycle de changement d'état est effectué à la réception d'un événement *move*. Les composants émettent en fin de cycle respectivement un événement *speaked* et *moved* à destination du gestionnaire de simulation. Un cycle complet est composé d'un cycle de communication suivi d'un cycle de changement d'état.

Un cycle partiel est réputé terminé lorsque tous les composants ont émis leur événement de fin de cycle, ce qui autorise le gestionnaire de simulation à déclencher le cycle partiel suivant. Au cours d'un cycle, tous les composants évoluent de manière asynchrone entre eux.

## 2.2 Gestion des entrées/sorties - Marquage et démarquage d'une place d'entrée ou d'une place de sortie

Chaque composant atomique DEVS génère un réseau de Petri indépendant (voir figure 2b). Lorsque les entrées et les sorties sont connectées dans le modèle DEVS, comme sur la figure 2a, les places correspondantes du réseau de Petri doivent être connectées. Dans le cas d'une exécution sur un même nœud, les places de sortie d'un composant sont fusionnées avec les places d'entrée qui lui sont reliées par des connecteurs (figure 2c). En revanche, dans le cas d'une exécution distribuée, il est impossible de réaliser directement une fusion.

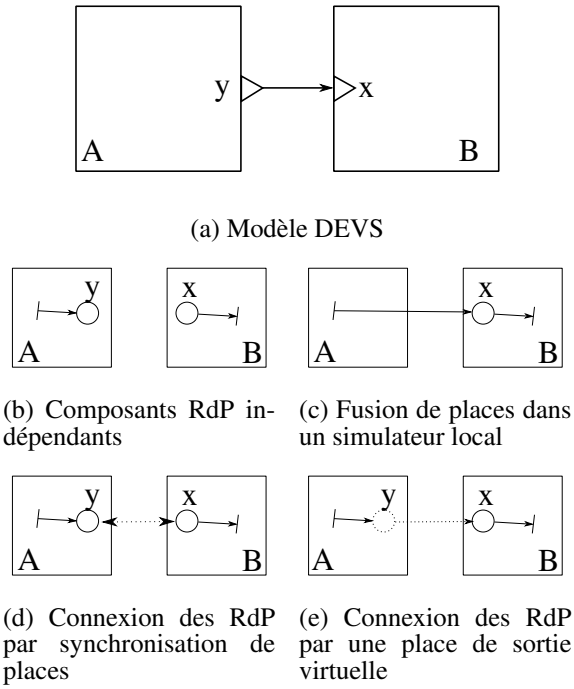
La première solution pour répondre à cette pro-

blématique consiste à conserver une place de sortie dans le composant source et une place d'entrée dans le composant cible, et de mettre celles-ci en correspondance (figure 2d). Ainsi, une évolution du marquage de la place de sortie déclenche une procédure de communication qui va venir mettre à jour le marquage de la place d'entrée, et vice-versa.

Néanmoins, cette approche peut être optimisée. En effet, on remarque que par construction du réseau de Petri individuel correspondant à un composant atomique DEVS, les places de sortie ne sont jamais relues à l'intérieur du composant émetteur. Il est donc inutile de mettre à jour la place de sortie lors du démarquage de celle-ci, car son état ne sera pas utilisé. Par extension, l'état de la place de sortie elle-même n'a pas à être mémorisé. Il n'est donc nécessaire que de transférer *l'information* de marquage de la place de sortie vers celle d'entrée, et non d'avoir une place de sortie à proprement parler. La place de sortie est donc une place virtuelle, un simple proxy pour la connexion avec la place d'entrée correspondante, dont le marquage déclenche une action de communication visant à marquer la place d'entrée distante (figure 2e). Dans le cas où la sortie est connectée à plusieurs entrées, l'information devra être communiquée à toutes les places en question.

Ce fonctionnement est équivalent à un mécanisme de fusion sur les entrées, car la place de sortie disparaît au profit d'une communication. Cette méthode fournit un résultat équivalent à l'approche présentée dans [6] pour la séparation d'un RdP en sous-réseaux distribués sur des nœuds de simulation indépendants, dans laquelle la communication est réalisée sur l'arc ayant pour source une transition et pour cible une place.

Deux informations sont à transmettre lors d'une communication DEVS : l'événement et la valeur de la variable associée. L'événement DEVS correspond au marquage de la place d'entrée. La valeur de la sortie doit donc également être communiquée au composant distant.




(a) Modèle DEVS

(b) Composants RdP indépendants

(c) Fusion de places dans un simulateur local

(d) Connexion des RdP par synchronisation de places

(e) Connexion des RdP par une place de sortie virtuelle

Figure 2 – Connexion de composants DEVS

Ainsi, le protocole de communication qui remplace le marquage d'une place de sortie prend deux paramètres : la référence de la place de sortie en question afin de permettre d'établir la correspondance avec les places d'entrée connectées, ainsi que la valeur de la variable de sortie à l'instant de l'événement.

### 2.3 Gestion du temps du prochain événement

La synchronisation du temps du prochain événement entre les composants distribués doit se faire à chaque pas de simulation, correspondant à un cycle complet. Pour cela, chaque composant indique en fin de cyle, avec son événement *moved*, son propre temps d'imminence $t_n$, et le minimum entre tous les $t_n$ est choisi en tant que temps du prochain événement. Ainsi, en fin de cycle, chaque composant DEVS retourne son $t_n$ au gestionnaire de simulation, qui effectue un calcul du minimum $t_{n_{min}}$ et retransmet celui-ci à chaque composant au début du prochain cycle, avec l'événement *speak*, afin que ceux-ci déterminent s'ils sont ou non imminents.

### 2.4 Détail d'un scénario de simulation

Sur la figure 3, on présente un scénario de simulation distribuée entre deux simulateurs partiels. Chaque simulateur partiel est constitué, comme indiqué précédemment, de deux threads qui communiquent de manière asynchrone. La boucle principale consiste en une série de « doStep » répétés jusqu'à ce que le simulateur atteigne le temps voulu.

La première phase présente l'exécution du cycle de communication débuté par l'émission de l'événement *speak*. Au cours de ce cycle partiel, un message est émis depuis le premier simulateur. Le second simulateur, n'ayant pas de message à émettre, retourne directement l'événement *speaked*. En revanche, le premier simulateur, après émission de son message à destination du second, se met en attente de la notification *ack* indiquant que le message a bien été transmis, ce qui signifie que la place d'entrée distante a bien été marquée. Une fois cette notification reçue, le simulateur peut terminer son cycle et émettre son événement *speaked*. Le gestionnaire de simulation ayant reçu tous les événements de fin de cycle de communication passe alors au cycle de changement d'état par l'émission de l'événement *move*, qui se termine lorsque tous les événements *moved* sont reçus par le gestionnaire.

La transmission du marquage des places d'entrées décrite précédemment est ici mise en œuvre par les appels *readValue* et *writeValue*. L'attente de la confirmation de marquage de place par le réseau de Petri qui émet ce type d'événement assure que le message est bien reçu par le simulateur partiel distant avant la fin du cycle de communication.

## 3 Mise en œuvre

Nous considérons la classe des réseaux de Petri temporisés telle que définie dans [10], où un intervalle de temps est associé aux transitions. Notons que le codage que nous avons réalisé ne contient que des transitions à intervalle de temps ponctuel, c'est à dire sous la forme $[\theta; \theta]$.

Nous avons réalisé une application logicielle et une application matérielle qui intègrent la description complète du réseau de Petri : structure de contrôle (places et transitions) et structure de données (prédicats et données).

Notre mise en œuvre ne respecte pas les règles de fonctionnement du modèle formel mais la structure de codage que nous avons définie assure la préservation de la sémantique du modèle. En effet, dans un réseau de Petri, il n'est normalement possible de tirer qu'une seule transition à la fois. Or, pour cette mise en œuvre, nous avons relaxé cette règle pour autoriser le tir simultané de plusieurs transitions indépendantes dans des composants différents. La prossibilité de prendre cette liberté par rapport au modèle des RdP pour des transitions indépendantes est démontrée dans [6] dans leur proposition d'un simulateur distribué de RdP.

Dans notre cas, nous avons démontré, tel que spécifié dans [7], que le tir simultané de plusieurs transitions sensibilisées à partir d'un marquage initial donné conduit au même marquage final que celui obtenu en tirant successivement chacune de ces transitions sensibilisées. Cette relaxe nous permet d'une part d'optimiser les performances par le tir simultané de plusieurs transitions et d'autre part de simplifier considérablement la mise en œuvre.

### 3.1 Mise en œuvre logicielle

Pour la mise en œuvre logicielle, nous avons utilisé une évolution dirigée par les transitions. Cette mise en œuvre et d'autres, pour les réseaux de Petri non temporisés, sont détaillées dans [7]. Dans cette mise en œuvre, le réseau est envisagé par l'évaluation de la sensibilité des transitions, en faisant évoluer le marquage des places lors du tir de l'une des transitions.

Si le marquage de toutes les places amonts d'une transition est vrai et que sa condition logique associée est vraie, la transition est sensibilisée. Si une transition est nouvellement sen-

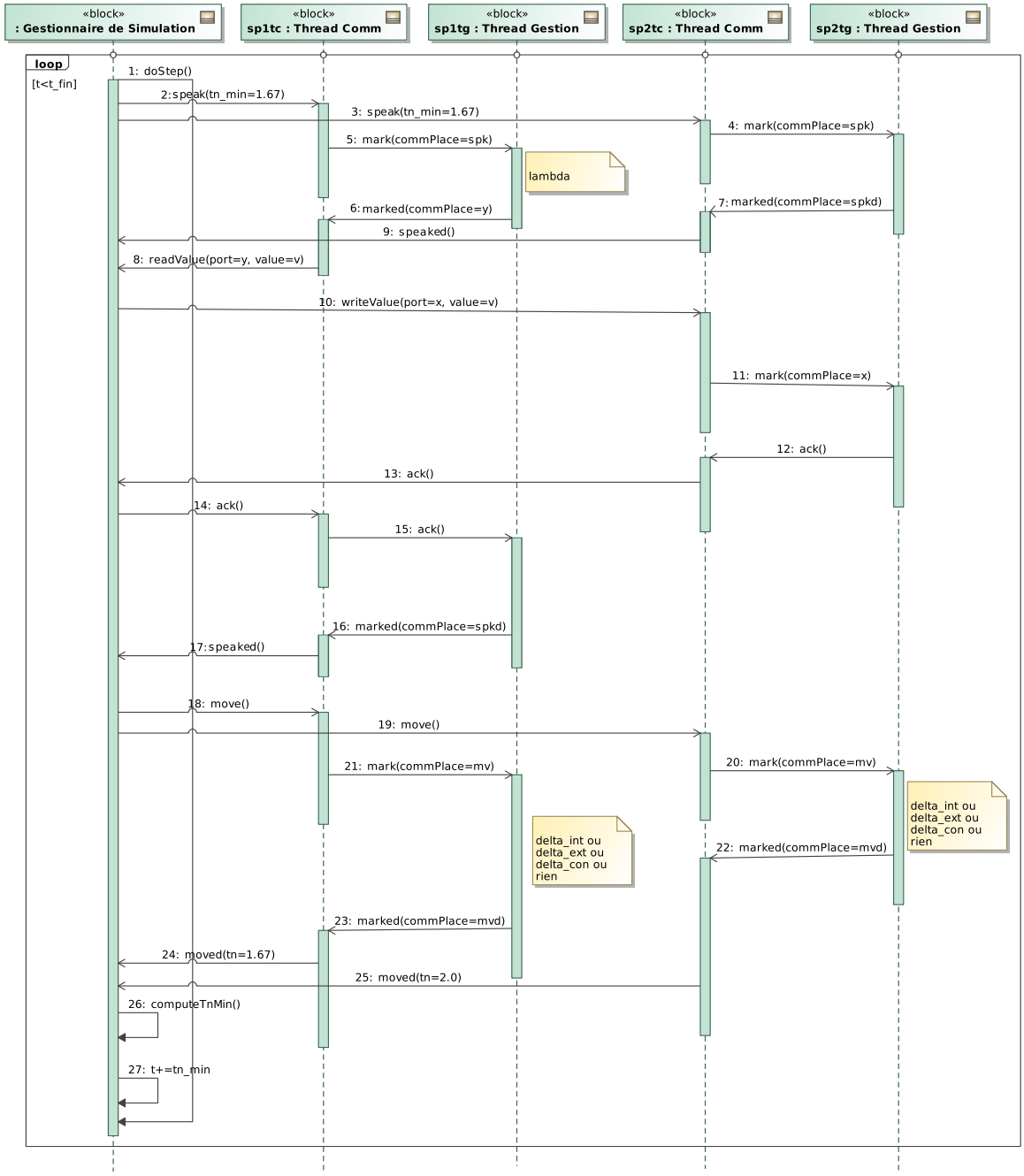



Figure 3 – Scénario de simulation

sibilisée alors son intervalle de temps prend sa valeur initiale, si une transition est sensibilisée et qu'elle était sensibilisée au cycle précédent alors on y soustrait le plus petit intervalle de toutes les transitions sensibilisées au cycle précédent. Ensuite on évalue si une transition est franchissable. Une transition sensibilisée est franchissable si son intervalle de temps est le plus petit des intervalles de temps de toutes les transitions sensibilisées. Dans ce cas la transition est dite imminente et est franchissable. Le franchissement d'une transition fait évoluer le marquage des places en amont et en aval de la transition et les valeurs des actions sont calculées.

**Structure de contrôle**

— $marked$, $mark$ et $unmark$ sont des vecteurs de booléens, un composant par

place du réseau,
— $enabled$, $newly\_enabled$, $imminent$, $fireable$ et $fired$ sont des vecteurs de booléens, un composant par transition du réseau,
— $interval$ et $init\_interval$ sont des vecteurs de réels, un composant par transition du réseau.

Toutes les places qui sont fusionnées au sein du même simulateur partiel sont codées par une mémoire partagée.

### 3.2 Mise en oeuvre materielle

En VHDL, la plateforme cible est la carte de développement ZedBoard. Celle-ci est architecturée autour d'un FPGA Zynq de Xilinx, FPGA contenant en « dur » un processeur ARM Cortex A9.

Afin de mettre en œuvre la communication avec le gestionnaire de simulation, notre code VHDL est encapsulé dans un wrapper de bus AXI, qui permet une communication directe avec le processeur. Sur le processeur, un simple programme multithreadé reposant sur l'OS FreeRTOS doté de la bibliothèque LwIP supervise l'état du réseau de Petri afin de remonter les informations nécessaires vers le gestionnaire de simulation, et réceptionne les ordres de celui-ci concernant l'évolution du réseau de Petri. La séparation en threads présentée plus haut est ici accompagnée d'une séparation « physique », la communication avec le gestionnaire de simulation étant réalisée sur le processeur tandis que l'exécution du réseau de Petri est elle réalisée dans la partie reconfigurable du FPGA.

Les différents threads du programme FreeRTOS comprennent principalement :
— la gestion de la communication avec le gestionnaire de simulation,
— la surveillance de l'état du réseau de Petri matériel et de ses sorties,
— d'éventuels plug-ins.

Les plug-ins utilisent un mécanisme de branchement par appel de fonction, recevant les informations d'évolution des sorties du réseau de Petri. Ceci nous permet par exemple de brancher un afficheur à LED pour visualiser les sorties d'un modèle de jeu de la vie.

La mise en œuvre du réseau de Petri lui-même en VHDL est constituée de composants indépendants représentant les différents modèles atomiques, et communiquant entre eux. La structure de contrôle est composée de places synchrones à l'horloge de simulation, et de transitions combinatoires. Les places sont représentées par des bascules à deux états, marquées ou non, stimulées par les transitions qui viennent activer les entrées de marquage et de démarquage. Les transitions, combinatoires, sont asynchrones et dépendent de l'état des places amont ainsi que de la temporisation qui leur est associée. La vitesse de l'horloge de simulation doit être telle que la stabilisation de la partie combinatoire représentant les transitions ait lieu avant le prochain cycle.

Le point clé ici est la gestion de la temporisation des transitions. Nous avons choisi de déplacer celle-ci en dehors de la transition elle-même, afin de centraliser le calcul dans un gestionnaire de temps. Le gestionnaire de temps connaît toutes les transitions présentes dans le réseau de Petri local, ainsi que le temps $t_{n_{min}}$. Ceci lui permet de déterminer à tout moment les transitions temporisées qui doivent être sensibilisées.

## 4 Conclusion et perspectives

Nous avons présenté les concepts d'une architecture de simulation distribuée de modèles DEVS basée sur des réseaux de Petri temporisés qui communiquent par des places au travers d'une connexion de type socket TCP. Dans cette architecture, les plateformes qui accueillent la mise en œuvre des réseaux de Petri peuvent être logicielles, matérielles ou mixtes. Ce travail est en cours de développement. Il n'est pas encore intégré à la version publique de l'outil de simulation DEVS (voir [2]), mais nous avons des bases solides pour les développements restants.

Une étude des performances de notre architecture distribuée devra être réalisée. Nous avons déjà obtenu des résultats significatifs sur une mise en œuvre de la simulation intégralement sur FPGA, toutefois ceux-ci sont fortement impactés par les nécessaires communications avec l'interface logicielle. Un travail sur l'optimisation des communications et des synchonisations du modèle RdP reste donc à réaliser pour la version distribuée. Pour cela, l'utilisation de nouvelles extensions de RdP telles [11], qui propose une version nativement distribuée des RdP temporisés fera l'objet d'une étude.

En effet, nous prévoyons d'étendre notre approche à la simulation en temps réel pour le couplage des modèles à des procédés réels par l'implémentation de mécanismes de synchronisation entre la mise en œuvre proposée et les entrées/sorties réelles. La gestion distribuée de la simulation permet notamment de remplacer un modèle par un composant physique, pour peu que celui-ci dispose de la même interface que le modèle.

D'autre part nous prévoyons d'encapsuler certaines opérations que nous avons mises en place pour la fabrication de composants compatibles avec FMI (Fonctional Mockup Interface), un standard de cosimulation et d'échange de modèle. Nous avons notamment déjà intégré la fonctionnalité d'import de FMU (Fonctional Mockup Unit) dans notre outil.

## Références


[1] Vincent ALBERT et Clément FOUCHER. « Formal Framework for Discrete-Event Simulation ». In : *20th IFAC World Congress*. Toulouse, France, juil. 2017.

[2] Vincent ALBERT et Clément FOUCHER. *Site web de Project DEVS*. URL : `https : / / www . laas . fr / projects / prodevs/` (visité le 21/02/2018).

[3] Vincent ALBERT et Sangeeth PONNUSAMY. « Codage de CDEVS et de PDEVS en réseau de Petri temporisé ». In : *Les journées DEVS Francophones (JDF)*. Cargèse, France, avr. 2016, p. 67-75.

[4] Benjamin CAMUS et al. *MECSYCO : a Multi-agent DEVS Wrapping Platform for the Co-simulation of Complex Systems*. Rapp. tech. LORIA, UMR 7503, Université de Lorraine, CNRS, Vandoeuvre-lès-Nancy ; Inria Nancy, sept. 2016.

[5] Ghassan CHEHAIBAR. « Méthodes d'analyse hiérarchique des réseaux de Petri ». Thèse de doct. Ecole Nationale des Ponts et Chaussées, sept. 1991.

[6] Giovanni CHIOLA et Alois FERSCHA. « Distributed Simulation of Petri Nets ». In : *IEEE Concurrency* 1 (août 1993), p. 33-50. ISSN : 1092-3063.

[7] Michel COMBACAU, Philippe ESTEBAN et Alexandre NKETSA. « Commandes à réseaux de Petri Mise en œuvre et application ». In : *Techniques de l'ingénieur Automatique séquentielle* (2005).

[8] Robert KEWLEY, Niel KESTER et Joseph MCDONNELL. « DEVS Distributed Modeling Framework - A parallel DEVS implementation via microservices ». In : *2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*. Avr. 2016.

[9] Yong Jae KIM, Jae Hyun KIM et Tag Gon KIM. « Heterogeneous Simulation Framework Using DEVS BUS ». In : *SIMULATION* 79.1 (2003), p. 3-18.

[10] Tadao MURATA. « Petri Nets : Properties, Analysis and Applications. » In : *Proceedings of the IEEE* 77.4 (avr. 1989), p. 541-580.

[11] Mogens NIELSEN, Vladimiro SASSONE et Jiří SRBA. « Towards a Notion of Distributed Time for Petri Nets ». In : *Applications and Theory of Petri Nets 2001*. Springer Berlin Heidelberg, 2001, p. 23-31. ISBN : 978-3-540-45740-4.