UNIVERSITÉ DE MONTRÉAL

CONTINUOUS/DISCRETE CO-SIMULATION INTERFACES FROM FORMALIZATION TO IMPLEMENTATION

LUIZA GHEORGHE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE
PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR (Ph.D.)
(GÉNIE INFORMATIQUE)
AOÛT 2009

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

CONTINUOUS/DISCRETE CO-SIMULATION INTERFACES
FROM FORMALIZATION TO IMPLEMENTATION

présentée par: GHEORGHE Luiza
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment accepté par le jury d'examen constitué de :

M. QUINTERO Alejandro, Doct., président
Mme. NICOLESCU Gabriela, Doct., membre et directrice de recherche
M. AIT MOHAMED Otmane, Ph. D., membre
M. VANGHELUWE Hans, Ph. D., membre

DEDICATION

To my family

ACKNOWLEDGMENTS

There are many generous people that made this work possible and to whom I want to express my most heartfelt appreciation

My warmest thanks go to Mrs. Gabriela Nicolescu, my advisor, for her trust and support during all these years. To her I owe a great debt of gratitude for the advices, the confidence and the encouragement that helped me along the way. She has been a model adviser, patient and genuinely collaborative. Without her example I would not be where I am today.

Special thanks go to Mrs. Hanifa Boucheneb for her help, her advices and her selfless availability. Thank you for helping me discover the world of abstractions and formal representations.

I want to thank to Mr. Alejandro Quintero, Mr. Otmane Ait Mohammed, Mr. Hans Vangheluwe and Mr. Christian Cardinal for accepting to be part of the jury.

I also want to thank the ones that believed in me and helped me fulfill one of my dreams, that of teaching; Mrs. Gabriela Nicolescu, Mr. Guillaume-Alexandre Bilodeau and Mr. Pierre Langlois my most sincere appreciation

I would also like to thank Mr. Ahmed Amine Jerraya, at the time head of SLS at TIMA laboratory, for welcoming me in their group. Even though the time spent there was very short, it was rich in information and knowledge.

A very special note of gratitude goes to Mr. Ian O'Connor – his work, his advancements brought me to discover and helped me become interested in optical networks on chip.

The "Conseil de recherches en sciences naturelles et en genie du Canada" (CRSNG) was very generous in providing me with scholarship assistance – thank you.

I want to mention my lab colleagues with whom I shared the lab and with whom I worked with on different projects: Maimouna, Bruno, Alain, Taieb, Sebastien, Matthieu, Essaid, Youcef and Faouzi. Thank you! It was a great privilege to have met and worked with you and I hope our paths will meet in the future professionally and otherwise.

I would also like to thank my friends Ana, Vinnie and Joachim for understanding my busy working schedule and therefore not being able to meet them when we wanted. Thank you, guys!

And last but not least I would like to thank my family for putting up with me: my son Alex, for being, my husband Michael for being so understanding and my mother who gave me the discipline to go through difficult times, all of them have been very supportive of my studies and my work. I only wish my father was still with us and could be happy for me, however his life is always a beacon for me now and in the future, may he rest in peace.

Thank you all and I sincerely apologize to anyone I have left out or omitted inadvertently.

ABSTRACT

Today's systems-on-chip are growing in complexity as a result of a higher density of components on the same chip, and also on account of the heterogeneity of different modules that are particular to different application domains (i.e. mechanical, electrical, optical, biological and chemical). These systems can be found in a broad and diverse spectrum of applications in many industries, including but not limited to Automotive, Aerospace, Health Care and, Consumer Electronics. These multi-domain heterogeneous systems enable new applications and the creation of new markets. This thesis focuses on the design and the simulation of heterogeneous embedded systems, more specifically on continuous/discrete heterogeneous systems.

Continuous-time and discrete-event models are at the core of the design of multi-domain systems. We present here a generic, language independent methodology for the design of continuous/discrete heterogeneous systems. This methodology is the basis for design of a new framework providing the interfaces that are in charge with the heterogeneous components adaptation. The methodology was successfully used for the implementation of different continuous/discrete systems such as: a glycemia level regulator, an analog/digital converter, a PID controller, a production chain control system and wimax system.

Parts of the proposed methodology were adapted for the formalization, modeling and verification of an optical network on chip.

RÉSUMÉ

Les systèmes sur puce sont de plus en plus complexes, pas seulement en terme de densité de composants sur la même puce mais aussi en terme d'hétérogénéité des modules spécifiques pour différents domaines d'application (mécanique, électrique, optique, biologique chimique). On retrouve ces systèmes dans un grand éventail d'applications et dans divers industries tels que l'automobile, l'aéronautique, la santé, l'électroniques et autres. Ces systèmes hétérogènes multi-domaine permettent de nouvelles applications et la création de nouveaux marchés. Cette thèse se concentre sur la conception et la simulation des systèmes hétérogènes embarqués.

Les modèles temps-continu et événement discret sont le noyau de la conception des systèmes multi-domaine. On présente ici l'analyse de modèles d'exécution et modèles de synchronisation des systèmes hétérogènes continu/discret, la définition d'une méthodologie générique pour la conception des outils de co-simulation des systèmes hétérogènes continus/discrets et la validation de la méthodologie par applications – la réalisation d'un cadre de co-simulation pour les systèmes continu/discret. La méthodologie exploite les techniques de vérification formelle et de la simulation. La conception des outils de simulation est basée sur la définition d'une architecture générique des interfaces de simulation ainsi que sur des modèles de synchronisation vérifiés formellement. La méthodologie a été utilisée pour l'implémentation d'un régulateur de niveau de glycémie. Une partie de la méthodologie a été adaptée pour la formalisation, la modélisation et la vérification formelle d'un réseau optique sur puce.

CONDENSÉ EN FRANÇAIS

Les systèmes sur puce sont de plus en plus complexes, pas seulement en terme de densité de composants sur la même puce mais aussi en terme d'hétérogénéité des modules spécifiques pour différents domaines d'application (mécanique, électrique, optique, biologique chimique). On retrouve ces systèmes dans un grand éventail d'applications et dans divers industries tels que l'automobile, l'aéronautique, le médical, l'électroniques et autres. Ces systèmes hétérogènes multi-domaine permettent de nouvelles applications et la création de nouveaux marchés. Les modèles temps-continu et événement discret sont le noyau de la conception des systèmes multi-domaine. Ce projet s'articule autour d'un point clé pour la conception des systèmes continus/discrets (C/D): la conception à partir d'un niveau haut d'abstraction. Le projet propose une méthodologie indépendante des langages de programmation qui permet la conception efficace des outils de co-simulation pour tels systèmes. La méthodologie a été utilisée pour la conception d'un nouveau cadre qui fournit des interfaces en charge avec l'adaptation des composants hétérogènes. Ce cadre a été utilisé pour l'implémentation d'un régulateur de niveau de glycémie.

1. Problématique

L'intégration des composants hétérogènes à un niveau élevé d'abstraction nécessite un nouveau cadre conceptuel pour l'abstraction des différentes interfaces qui réalisent l'adaptation entre les composants hétérogènes ainsi que des nouvelles méthodologies pour la validation. L'hétérogénéité implique l'utilisation des modèles en temps continu ainsi que des modèles à événements discrets dans un modèle globale, donnant une vue d'ensemble du système. Étant donnée l'hétérogénéité des concepts manipulés par ces deux types de modèles, la validation globale demande des interfaces de simulation capables de fournir des modèles de synchronisation qui peuvent accommoder le domaine continu et le domaine discret. Dans le cas des outils de validation plusieurs sémantiques d'exécution doivent être prises en considération pour réaliser la simulation

globale. La technique de validation la plus souvent utilisée est la co-simulation. La co-simulation permet l'exécution concurrente des différents simulateurs en parallèle. Cette validation élimine la détection tardive des erreurs et réduit le temps de conception. Il est donc nécessaire de définir un modèle d'exécution globale dont les éléments de base sont :

- les *modèles* des composants du système hétérogène qui sont décrits en temps continu ou bien dans le domaine des événements discrets
- les *interfaces de co-simulation* qui réalisent l'adaptation de chaque modèle au bus de co-simulation, l'adaptation des différents protocoles de communication et la synchronisation entre les deux modèles.
- le *bus de co-simulation* qui est responsable de l'interprétation des interconnections entre les deux modèles composant le modèle global.

Les aspects qui rendent difficile la modélisation et la simulation des systèmes continus/discrets sont [6]:

- pour le modèle discret *le temps* est une notion globale pour tous les modules du système, il avance discrètement en passant par les instants discrets définis par les temps de notification des événements discrets. Pour le modèle continu le temps est une variable globale qui avance par le temps d'intégration (continu ou variable);
- pour le modèle discret *les processus* sont sensibles aux événements alors que, pour le modèle continu, les processus sont exécutés à chaque pas d'intégration;
- pour le modèle discret *la communication* est réalisée par ensembles d'événements alors que pour le modèle continu, la communication est réalisée par des signaux continus (un signal continu possède une valeur à touts instants);
- chaque modèle doit être capable de *détecter*, de *localiser* en temps et de *réagir* aux *événements* envoyés par l'autre modèle.

La conception des interfaces de co-simulation est couteuse en termes de temps, est une source d'erreurs, est difficile à déboguer, influence les performances de la simulation

globale et demande la compréhension exhaustive des simulateurs impliqués dans la co-simulation. La clé de voûte consiste donc en la définition rigoureuse du comportement et de l'architecture des interfaces de co-simulation pour la génération automatique.
Le modèle formel qui est la représentation abstraite, rigoureuse d'un modèle, représente la base de la définition d'un outil générique de co-simulation qui fournit des modèles globaux de co-simulation pour la validation des systèmes hétérogènes continus/discrets. En représentant le modèle formel tous les requis du système sont précisément définis et toutes les inconsistances et les ambigüités sont éliminées.

1.1 Objectives et contributions

Les objectifs du travail présenté ici sont :

- la définition d'une approche pour la conception des outils de validation efficaces pour les systèmes hétérogènes
- l'intégration, dans l'étape de validation, de nouveaux aspects spécifiques pour la nouvelle génération de systèmes hétérogènes multi-domaine : l'interaction entre le modèle continu et le modèle discret

Les contributions plus spécifiques sont :

- l'analyse de modèles d'exécution et modèles de synchronisation des systèmes hétérogènes continu/discret
- la définition d'une méthodologie générique pour la conception des outils de co-simulation des systèmes hétérogènes continus/discrets.
- la validation de la méthodologie par applications – la réalisation d'un cadre de co-simulation pour les systèmes continu/discret, l'implémentation d'un régulateur de glycémie et la modélisation et la vérification formelle d'un réseau optique passif, sur puce.

2. Revue de littérature

Cette section est un survol de travaux existants. Ces travaux peuvent être divisés en deux catégories : une basée sur la simulation et une basée sur la représentation formelle.
Dans la première catégorie il existe deux approches pour réaliser la co-simulation des systèmes hétérogènes : une approche homogène et une approche hétérogène ([5]):

- L'approche *homogène* où les concepteurs utilisent un seul langage pour la spécification complète du fonctionnement du système et donc les descriptions des diverses parties sont réalisés dans un langage unique de simulation (tel que le C pour accélérer les simulations) ([9], [10], [11],, [12] [13], [14], [15]) La difficulté est d'être assuré que la traduction et la simulation du langage unique ne change pas la sémantique des descriptions des diverses parties.
- L'approche *hétérogène* ou les concepteurs utilisent des langages spécifiques pour la modélisation des différents modules d'un système complet et donc ils conservent les descriptions spécifiques des diverses parties et exécutent en parallèle les divers simulateurs ([18], [19], [20]). La communication et la synchronisation entre simulateurs sont assurées par le bus de co-simulation. Cette tache peut être difficile lorsque les modèles de simulation sont différents.

Ayant la description informelle du système, il est nécessaire d'avoir la description du modèle dans une forme abstraite de spécification à base de règles. Cette forme caractérise le modèle dans un langage mathématique, celui de la théorie des ensembles ou de la théorie des systèmes ou un autre paradigme formalisé [4].
Dans les domaines des définitions formelles et du formalisme, on peut énumérer les travaux de :

- l'Université de Berkeley [25] ou les auteurs proposent un cadre formel pour la comparaison de plusieurs MoCs;
- « Royal Institute of Technology » de Stockholm [26], [27]ou l'auteur a proposé un cadre formel qui sépare les aspects de communication /synchronisation et le

comportement interne. Un processus est divisé en deux parties : le noyau du processus qui est responsable du calcul et l'enveloppe (de l'anglais « shell ») du processus en charge de la communication avec les autres processus;

- l'Université d'Arizona [28], [29] ou les auteurs définissent un formalisme mathématique DEVS pour la spécification d'un système. Ce formalisme est une représentation d'un système à « entrée-sortie » ayant une base de temps réel et continu. Des travaux sur les modèles ou le système discret retourne en arrière sont présentées dans [35] [36], [37] et [38].

Les travaux basés sur la simulation, l'approche homogène sont couteux en termes de temps de développent des nouvelles bibliothèques de composants et temps d'apprentissage pour les développeurs qui travaillent avec les outils. Dans le cas des travaux basés sur la simulation, l'approche hétérogène les interfaces sont conçues ad-hoc, ne sont pas vérifies formellement et ne se concentrent pas sur les domaines continu et discret. Cette thèse propose une approche ou les développeurs travaillent avec des outils très populaires et peuvent réutiliser des modèles qui existent dans des bibliothèques déjà testées. Les interfaces sont vérifiées formellement et sont générées automatiquement.

Les travaux basés sur la représentation formelle fournissent une base abstraite pour les systèmes hétérogènes mais ils ne prennent pas en considération les interfaces de co-simulation ou ils ne permettent pas la vérification formelle. Cette thèse se concentre sur les interfaces de co-simulation et donne un mécanisme pour la représentation formelle et la vérification formelle des interfaces de co-simulation,

3. Concepts de base

Cette section introduit les concepts de base qui seront utilisés dans ce travail : les modèles d'exécution et les simulateurs continu et discret, le modèle de synchronisation ainsi que la définition de l'environnement de simulation continu/discret.

3.1 Modèles d'exécution

3.1.1 Modèle à événements discrets

La simulation d'un modèle purement discret est basée sur les événements, elle est généralement accomplie en utilisant un simulateur à événements discrets. Le rôle de simulateur est de maintenir l'ordre des événements dans une file d'attente suivant leur temps de notification. A chaque itération, le simulateur fait sortir de la file l'événement qui a le temps de notification le plus proche et exécute les processus sensibles à cet événement. L'exécution de ces processus peut générer d'autres événements entraînant l'exécution d'autre processus. Si les événements dont le temps de notification égale au temps actuel sont tous traités, le simulateur avance le temps pour le plus proche événement planifié.

3.1.2 Modèle en temps continu

La simulation d'un modèle purement continu régi par des équations différentielles et algébriques est basée sur la résolution numérique de ces équations. La plupart des algorithmes de résolution discrétisent le temps en un ensemble d'instants. La différence entre deux instants est appelée *pas d'intégration* ou pas de calcul et suivant l'algorithme utilisé ce pas peut être fixe ou variable. Les critères utilisés pour le choix du pas d'intégration sont : la précision, la stabilité et la continuité des signaux. Dans les cas où la précision est la seule condition (le cas ou le modèle est stable et il n'y a pas de discontinuités), il est possible d'utiliser un algorithme à pas fixe. L'utilisation d'un algorithme à pas variable augmente les performances de simulation. Pour satisfaire les critères de précision l'algorithme réduit le pas quand le modèle évolue rapidement. Pour éviter les calculs qui ne sont pas nécessaires et améliorer la vitesse de simulation l'algorithme augmente le pas quand le modèle évolue lentement,.

Pour une synchronisation rigoureuse, chaque simulateur impliqué dans la co-simulation C/D doit considérer les événements provenant de l'environnement externe. Ils doivent

s'arrêter avec précision aux échantillons de temps de ces événements (détection des événements). Ces échantillons de temps sont des points de communication entre les deux simulateurs.

Le simulateur continu doit détecter le prochain événement discret planifié par le simulateur discret. Cette détection implique l'ajustement des pas d'intégration pour le simulateur continu. Le simulateur discret doit détecter les événements d'état. Un événement d'état est un événement non prédictible qui est généré par le simulateur continu et qui a une estampille de temps dépendante des valeurs des variables d'état (comme par exemple les événements « passage a zéro » ou le dépassement d'un seuil). La conséquence est le contrôle de l'avancement en temps des simulateurs discrets (au lieu d'avancer le pas de simulation prévu, le simulateur avance précisément jusqu'au moment de l'évènement d'état).

3.2 Modèles de synchronisation

Table 1. Modèles de synchronisation

Modèle de synchronisation	Pas de synchronisation	Avantages	Désavantages
Le modèle canonique	A chaque pas discret et chaque occurrence d'un événement	Général	Surdébit de synchronisation
Le modèle de synchronisation avec retour en arrière	A chaque occurrence des événements de mise à jour, événements d'échantillonnage et événements d'état	Evénements de mise à jour, événements d'échantillonnage non-périodiques, efficace si le modèle continu ne génère pas des événements d'état	Retour en arrière pour le modèle discret est requis ci le modèle continu génère des événements d'état.

Ce projet est basé sur deux modèles de synchronisation :

- le modèle canonique – ou le simulateur continu avance le temps avant le simulateur discret.
- le modèle de synchronisation avec retour en arrière (« rollback » en anglais) - ou le simulateur discret avance le temps avant le simulateur continu.

Le Table 1 montre les deux modèles de synchronisation comparés de point de vue pas de synchronisation ainsi que leurs avantages et désavantages.

4. Méthodologie de conception des outils de co-simulation

Cette section présente notre approche pour la spécification et la simulation des systèmes hétérogènes continus/discrets. L'accent sera mis sur les interfaces de simulation et leur génération automatique, le bus de co-simulation, ainsi que sur la communication et la synchronisation entre le deux modèles.

Pour permettre la conception des outils de co-simulation, la méthodologie qu'on propose est formée de deux étapes indépendantes des outils de co-simulation utilisés pour simuler le modèle continu et le modèle discret (voir Figure 1). Pendant ces étapes les interfaces de co-simulation sont définies dans un cadre conceptuel, leurs fonctionnalités et l'architecture interne sont décrites à l'aide des formalismes existants et logique temporelle.

Les deux étapes sont:

1. L'étape générique incluant les tâches suivantes:
   - Définition de la sémantique opérationnelle des modèles de synchronisation pour le modèle global de co-simulation.
   - Distribution de la fonctionnalité de synchronisation entre les interfaces de co-simulation.
   - Formalisation et vérification du comportement des interfaces de co-simulation.

- Définition des éléments de la bibliothèque et l'architecture interne des interfaces de co-simulation.

2. L'étape d'implémentation incluant les tâches suivantes:

- L'analyse des outils de simulation pour les intégrer dans le cadre de co-simulation.
- L'implémentation des éléments spécifiques de la bibliothèque et validation de l'implémentation.

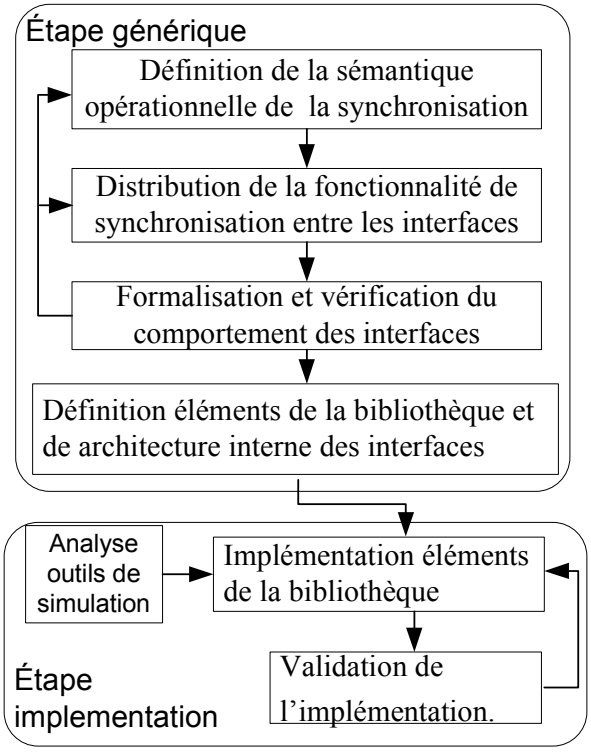


Figure 1. Méthodologie générique pour la conception des outils de co-simulation

Les tâches de l'étape générique sont détaillées dans les sous-sections suivantes.

4.1 Définition de la sémantique opérationnelle des modèles de synchronisation pour le modèle global de co-simulation

La sémantique opérationnelle est la représentation du comportement du système dans une forme mathématique, rigoureuse, non-ambigüe. Ce modèle sert comme base pour

l'analyse et la vérification. Dans notre travail, pour définir la sémantique opérationnelle, sous forme de règles, on a utilisée le formalisme proposé par [28], [29] – Discrete Event System Specifications (DEVS). Nous avons définit la sémantique opérationnelle des deux modèles de synchronisation présentés dans la section 3.2.

4.2 Distribution de la fonctionnalité de synchronisation entre les interfaces de co-simulation

Apres la sémantique opérationnelle, la fonctionnalité de la synchronisation est distribuée entre les interfaces de co-simulation. Le premier pas de cette opération consiste en l'identification de la sémantique opérationnelle de chaque interface, à partir de la sémantique globale. La fonctionnalité de chaque interface a été par la suite modélisée à l'aide des automates temporisés.

4.3 Formalisation et vérification du comportement des interfaces de co-simulation

La formalisation et vérification formelle des interfaces de co-simulation peut être divisée en trois pas : la représentation formelle, la simulation du modèle formel et la vérification formelle. Pour réaliser cette étape on a utilisé les automates temporisés ([46], [47]) et l'outil UPPAAL ([48]).

4.4 Définition des éléments de la bibliothèque et l'architecture interne des interfaces de co-simulation

La vérification formelle du comportement des interfaces est suivie par la définition de l'architecture interne des interfaces de co-simulation. Cette définition est une étape clé pour la génération automatique des interfaces de co-simulation. Dans notre approche les interfaces ont été représentées comme un ensemble de modules hiérarchiques, en se basant sur des composants atomiques qui sont des éléments de la bibliothèque utilisée pour la génération automatique.

4.5 L'analyse des outils de simulation pour les intégrer dans le cadre de co-simulation

Des fonctionnalités spécifiques sont demandées pour les simulateurs continu et discret et donc l'intégration des outils de simulation dans l'environnement de co-simulation demande l'analyse des outils de simulation. Le simulateur continu doit détecter les événements d'état, il doit envoyer des données pour la synchronisation vers le modèle discret, permette des points d'interruption pendant la résolution des équations différentielles et la mise à jour des points d'interruption. Le modèle discret doit détecter la fin du cycle de simulation, doit permettre l'ajout/l'extraction de nouveaux éventements dans/de la queue de l'ordonnanceur et doit envoyer les résultats du traitement des données et l'information pour la synchronisation vers le simulateur continu.

4.6 L'implémentation des éléments spécifiques de la bibliothèque et validation de l'implémentation

Le dernier pas de la méthodologie est l'implémentation des éléments spécifiques de la bibliothèque et validation de l'implémentation. Cette étape dépend des outils de simulation choisis dans l'étape précédente, l'analyse des outils de simulation.

5. Résultats

A partir de la méthodologie présentée dans la section 3, l'outil de de co-simulation CODIS a été créé. Cet outil permet la modélisation et la simulation précise d'un système continu/discret. Les entrées dans le flot de simulation sont :

- le modèle continu (en Simulink® [16]) qui est schématique
- le modèle discret (en SystemC [8]) qui est textuel.

La sortie du flot est le modèle global de simulation. Les interfaces de co-simulation sont automatiquement générées par un générateur des interfaces qui reçoit à l'entrée des

paramètres définis par utilisateur via un générateur de script. À la sortie on obtient le modèle discret avec ses interfaces de simulation. Plus des détails sur CODIS peuvent être trouvées dans l'annexe 2 ([51], [52]). L'outil CODIS a été utilisé pour valider plusieurs applications, parmi eux, un régulateur de niveau de glycémie. Un régulateur de niveau de glycémie est un système qui représente une alternative pratique au traitement classique du diabète de type 1. Une technique plus avancée est la thérapie par pompe, un traitement qui fournit au corps insuline ou glucose en se basant sur les valeurs en temps réel de la glycémie. Cette application consiste dans la simulation d'un régulateur de niveau de glycémie. Le système est formé par deux sous-systèmes – un sous-système discret qui contrôle l'injection et un sous-système continu qui modélise le système d'injections, le patient et l'assimilation de glucose et insuline dans le sang.

6. Réseaux optiques sur puce

Dans cette partie on présente des résultats ou le formalisme et la vérification formelle sont appliqués pour la formalisation, la modélisation et la vérification d'un système hétérogène, un réseau optique sur puce. Ces résultats qui sont des résultats complémentaires, ou une partie de la méthodologie proposée a été appliquée sont présentés dans l'annexe 1.

Les systèmes modernes sur puce intègrent plusieurs composants hétérogènes comme différents processeurs, composants matériel et interconnexions complexes qui utilisent différents protocoles de communication. Les interconnections sur puce sont limitatifs de point de vue performance et consommation d'énergie. La croissance, en termes de nombre, des composants intégrés sur une puce augmente l'impact des effets comme le bruit causé par la diaphonie, les interférences électromagnétiques qui peuvent produire des erreurs de données, les délais et autres [56]. Les réseaux optiques sur puce s'avèrent une solution intéressante. Parmi les avantages des réseaux optiques on peut mentionner : extensibilité, simplicité, surface réduite, guide d'ondes bidirectionnel, réduction de la diaphonie, charge capacitive, et de la distorsion du signal, débit élevé dans le guide

d'ondes. Les défis les plus importants sont l'accès au prototypage physique et la difficulté d'influencer les processus standard existants. Par conséquence, la modélisation et la simulation deviennent une alternative nécessaire pour l'exploration de ces systèmes. Plus des détails sur les concepts de base des réseaux optiques sur puce peuvent être trouvés dans l'annexe 2 de ce document.

Une partie de la méthodologie proposée dans ce travail a été utilisée pour aider les concepteurs à réaliser des modèles complexes de tels systèmes. Dans ce document on a proposé: la formalisation des interfaces opto-électriques a l'aide du formalisme DEVS[1], la formalisation des éléments passives de base composant un réseau optique[1] et la modélisation et la vérification formelle, pour la validation globale du comportement d'un réseau optique passif sur puce.

La modélisation et la vérification formelle ont été divisées en deux étapes. La première étape consiste dans la vérification d'un routeur $4 X 4 \lambda$ a un haut niveau d'abstraction et la deuxième étape a été la représentation du réseau a un bas niveau d'abstraction ou on a considéré seulement un initiateur et un chemin du signal a travers du réseau optique. On a vérifié les propriétés formelles pour les deux modèles. La vérification complète prend 2 secondes pour la première étape et 41 secondes/initiateur pour la deuxième étape.

7. Conclusions et perspectives

Cette thèse se concentre sur la conception et la simulation des systèmes hétérogènes embarqués, plus spécifiquement sur les systèmes multi-domaine ou plusieurs composants de différents domaines comme optique, électrique, mécanique, sont pris en considération.

---

1 Ce travail a été réalisé en collaboration avec Ph.D Mathieu Brière et Prof. Dr. Ian O'Connor, École Centrale de Lyon, France

7.1 Conclusion

Cette recherche a été motivée par le contexte courant des systèmes embarqués. On retrouve ces systèmes dans un grand éventail d'applications et dans divers industries tels que l'automobile, l'aéronautique, la santé, l'électroniques et autres. Ces systèmes hétérogènes multi-domaine permettent de nouvelles applications et la création de nouveaux marchés. Les modèles temps continu et événements discrets sont la base des systèmes multi-domaine. Ce travail cible les systèmes hétérogènes continu/discret, plus spécifiquement la conception d'un nouveau cadre qui fournit des interfaces de simulation en charge avec l'adaptation de divers simulateurs.

Un sommaire des contributions majeures est présenté ci-dessous :

- l'analyse des modèles d'exécution des systèmes continus et discrets et la définition des modèles d'exécution globaux basés sur deux modèles de synchronisation
- la définition d'une méthodologie générique pour la conception des outils de co-simulation des systèmes hétérogènes C/D. La méthodologie comporte deux étapes :
    - une étape générique ou la représentation des interfaces est raffinée d'un modèle de synchronisation abstraite jusqu'à l'architecture interne des interfaces de co-simulation
    - une étape d'implémentation
- la validation de la méthodologie par applications – la réalisation d'un cadre de co-simulation pour les systèmes C/D, l'implémentation d'un régulateur de glycémie et la modélisation et la vérification formelle d'un réseau optique passif, sur puce.

7.2 Perspectives

Cette thèse fait des progrès dans le développent d'une technique vérifiée pour la conception d'outils de co-simulation des systèmes hétérogènes continu/discret et ouvre des nouvelles directions pour les chercheurs qui travaillent dans la simulation au niveau système. La méthodologie proposée permet de nouveaux développements dans la génération automatique des interfaces de co-simulation pour les systèmes hétérogènes continu/discrets. Une nouvelle direction de recherche ouverte par ce travail est la vérification formelle de la composition des éléments de bibliothèque pour créer une interface. Une autre direction est l'analyse des modèles continus et discrets a intégrer pour vérifier la compatibilité en termes d'entrées, de sorties et de niveaux d'abstraction.

Ce travail peut être continué avec la modélisation et la simulation des systèmes hétérogènes C/D aux différents niveaux d'abstraction et l'intégration du modèle de synchronisation avec retour en arrière dans l'outil de co-simulation proposé. . Autres outils spécifiques pour le domaine discret peuvent être intégrés pour valider le travail (i.e SystemVerilog). Du travail peut être fait pour l'analyse de performance et l'optimisation des systèmes.

Un autre domaine dans lequel le travail présenté peut être exploité est la modélisation et la validation des réseaux optiques sur puce. Une direction pour les travaux futurs pourrait être l'intégration des composants optiques passifs et actifs avec des circuits intégrés, pour réaliser le modèle global d'exécution d'un réseau optique sur puce. A plus long terme, les interconnexions optiques peuvent être intégrées avec plusieurs processeurs sur la même puce et la méthodologie proposée peut être adaptée pour la modélisation et la validation d'un tel système.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

ABREVIATIONS

AMS	Analog mixed signal
API	Application programming interface
C/D	Continuous/Discrete
CDI	Continuous domain interface
CMOS	Complementary metal-oxide-semiconductor
CS	Continuous simulator
CSP	Communicating sequential processes
DDI	Discrete domain interface
DEVS	Discrete Event Specifications
DS	Discrete simulator
FSM	Finite state machine
HDL	Hardware description language
IP	Intellectual property
IPC	Inter process communication
ISS	Instruction set simulator
MoC	Models of computation
MEMS	Micro-electro-mechanical systems
MOEMS	Micro-opto-electro-mechanical systems
MPSoC	Multi-processors system on chip
ONoC	Optical network on chip
SDL	Specification and description language
SoC	System-on-chip
SOI	Silicone on insulator
VHDL	VHSIC (Very High Speed Integrated Circuits) hardware description language

LIST OF ANNEXES

INTRODUCTION

1. Heterogeneous Systems – Existing Context

System on chip (SoC) trends of the past decade observed the shrinking of the chips' size simultaneously with the growth in complexity. In response to the challenges of systems miniaturization, the International Technology Roadmap for Semiconductors (ITRS) emphasizes the More Than Moore's Law Movement that focuses on system integration rather than increasing transistor density and leads to a functional diversification in integrated systems [1]. Thus, system-on-chip are currently characterized by the heterogeneity of different modules that are particular to different application domains such as optical, electronical, mechanical, hydraulics and biological. These multi-domain systems are the main driver of the development of a wide range of products across a broad and diverse spectrum of applications in many industries, but not limited to Automotive, Aerospace, Health Care, Consumer Electronics, and others. These heterogeneous systems enable new applications and create new markets. ITRS states that heterogeneity is "a form of *diversity* that arises with respect to system-level SoC integration" and the design specification and validation are extremely challenging, particularly with respect to complex operating contexts [1].

Continuous-time and discrete-event models are at the core of the design of multi-domain systems. For instance Figure 2 gives an example of a glycemia level regulator that illustrates the above mentioned aspects. The electronics domain components can be found in this application in the control block. This block controls the injection of insulin and glucose. These injections are pumps, therefore they have mechanical fluidics components. The environment is the actual patient that is injected with insulin or glucose.

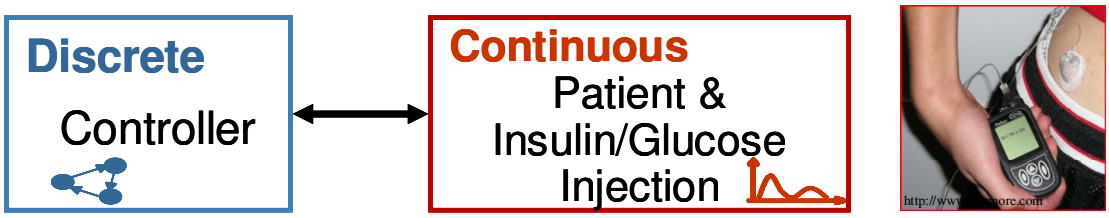


Figure 2. Glycemia level regulator

The control part is generally realized in the discrete domain using simulators for hardware and/or software components (i.e VHDL [6] or SystemC [8]).The patient, the pumps and the injection process are modeled in the continuous time domain using equations (an illustrative example is the utilization of a differential equation modeling the process of insulin assimilation in the human body). For the continuous components simulators integrating equation solvers are exploited (i.e. Simulink [16]).

2. Heterogeneous Systems - Problematic

The integration of continuous-discrete systems implies the cooperation between different teams with different cultures, using different specification languages and simulators. Given the diversity of concepts manipulated, the global design specification and the validation are extremely challenging; their heterogeneity makes more difficult the elaboration of a global execution model for the overall validation. Such a model may be very complex.

Currently, there are two techniques used for the validation of heterogeneous systems: the formal verification and the simulation.

In order to validate a system through formal verification, its behavior needs to be represented using a formal model. This representation has to clearly define the computation and the communication (and implicitly the synchronization) for the global model and verify the behavior of the interfaces. This approach has the advantage of a rigorous and unambiguous representation of the system's behavior. This allows for the exhaustive verification of an ensemble of system's properties. The challenge is however

the system's complexity that is difficult to manage and that has impact on the time and the cost.

The validation by simulation is currently the most popular validation technique and can be defined as the execution of a global representation of a heterogeneous system. The simulation was adopted by the designers for its advantages in terms of time invested for the validation and the facility of the utilization. However the choice of simulation has an incidence on the quality of the validation – it is well known that the simulation technique does not provide an exhaustive validation.

The simulation-based validation for heterogeneous systems is often referred as co-simulation. The co-simulation enables joint simulation of heterogeneous components with different execution models. Each heterogeneous component can be developed using a well known, domain-specific language and the resulting model can be reused later. The reusability advantages are: the development time, the time-to-market and the costs are reduced [3]. The co-simulation approach requires the elaboration of a global simulation model (Figure 3). The co-simulation interfaces have to provide efficient synchronization models for the adaptation of the domain specific models. This results in a complex behavior of the interfaces since their design is time consuming and a significant source of errors, they are difficult to debug and have impact on overall simulation performances. Moreover, co-simulation interfaces specification requires a deep understanding of the internal mechanism of the simulators involved in the co-simulation. Therefore, their automatic generation is very suitable.

New validation tools are required to facilitate the co-simulation during the design process. These tools generate automatically the global simulation model and consequently the co-simulation interfaces that adapt the heterogeneous models. The main role of these tools is to guarantee the correctness of the generated model, in order to accomplish this, the formal verification technique can be used during the co-simulation tools design.

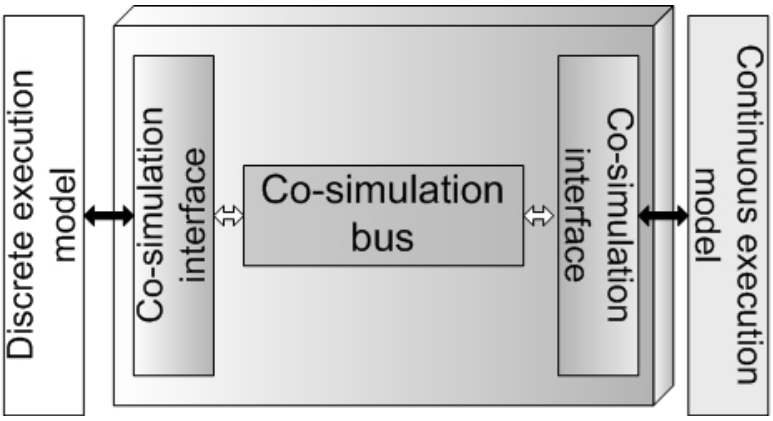


Figure 3. Global co-simulation model

3. Objectives and Contributions

The global objective of this thesis is the definition of new solutions that reduce the time and cost of the validation stage during the design flow of heterogeneous systems. The specific objectives are:

- The definition of an approach (technique, procedure) for the design of efficient validation tools for heterogeneous systems.
- The integration in the validation stage of new aspects specific to the next generation of multi-domain heterogeneous systems: the tight interaction between the continuous and discrete models.

The main contributions of this work are:

- The analysis of the continuous and discrete execution models and the definition of global continuous/discrete (C/D) execution models based on synchronization models.
- The definition of a generic methodology for the efficient design of C/D co-simulation tools. This methodology exploits the advantages of the formal verification-based and simulation-based validation techniques.
- The application of the methodology for the design of a validation tool.

These contributions are detailed in the next three sub-sections.

3.1 The Analysis of Continuous and Discrete Execution Models and Synchronization Models

The execution model can be viewed as the interpretation of a model of computation. In this work, we considered the continuous/discrete heterogeneous systems and their global execution model. Discrete and continuous systems are characterized by different physical properties and modeling paradigms. A global execution model has to take into account all these paradigms. As mentioned in the previous section, the elements that compose the global model are the execution models for the different components (the continuous execution model and the discrete execution model also called in this work simulators), the co-simulation interfaces and the co-simulation bus. In this thesis, the global execution model as well as the components execution models are analyzed. Moreover, the co-simulation interfaces have to provide efficient synchronization models for the modules adaptation. The two simulators have to detect, locate and retract events that are generated by the other simulator. While the discrete events are saved in a queue and their time stamp is already known, the continuous simulator can generate events at times that are not known beforehand (named here state events). The discrete simulator must react to these events. This requirement has to be accomplished by the synchronization. This thesis discusses two synchronization models: the canonical synchronization model and the rollback-based synchronization model.

3.2 The Definition of a Generic Methodology for the Efficient Design of Continuous/Discrete Co-Simulation Tools

This thesis proposes a methodology for the efficient design of continuous/discrete co-simulation tools. The methodology is composed of two main stages: a generic stage and an implementation stage. In order to enable the design of co-simulation tools, the generic stage of the methodology is refined through several steps that are independent of the simulation tools used for the continuous and discrete components of the system. During

these generic steps, the co-simulation interfaces are defined in a conceptual framework; their functionality and the internal structure of co-simulation interfaces are expressed using existing formalisms and temporal logic. After the rigorous definition of the required functionality for co-simulation interfaces, the designer will start the steps related to the implementation stage of the library elements using languages specific to different co-simulation tools. We emphasize here that the methodology is generic; the first stage is independent of the implementation languages of the co-simulation library.

3.3 Application of the Methodology to the Design of a Validation Tool

The proposed methodology was applied for the design of a co-simulation tool – CODIS (Continuous DIscrete Simulation) – that integrates the discrete simulator SystemC [8] and the continuous simulator Simulink® [16]. This tool was exploited for the validation of different heterogeneous systems such as glycemia level regulator, an analog/digital converter, a PID controller, a production chain control system and wimax system. In this thesis, we present the glycemia level regulator. Moreover, parts of the methodology were adapted for the formalization, the modeling and the validation of elements of an optical network on chip. This complementary work is presented in Annex 1.

4. Document Plan

This thesis is structured in five chapters, an introduction and a final section for conclusions and perspectives. Chapter 1 presents a survey of the existing works in the continuous/discrete heterogeneous systems modeling and validation. Both, formal verification-based and simulation-based approaches will be taken into consideration. Chapter 2 presents the basic concepts concerning the global execution model of continuous/discrete heterogeneous systems, their synchronization models and events update schema for the synchronization models. Chapter 3 introduces the methodology for the generation of global execution models. This chapter includes the validation of a

synchronization models with rollback (called here the rollback-based model) and without rollback (called here the canonical model). Chapter 4 shows the application of the methodology and the results of the implementation of a glycemia regulator where the continuous model was implemented in Simulink® and the discrete model was implemented in SystemC. Finally, the last section gives the conclusions. A part of the proposed methodology was applied for a passive optical network on chip and its implementation results are given in Annex 1 as complementary results.

CHAPTER 1. LITERATURE REVIEW

The existing works in the continuous/discrete systems validation field can be roughly divided into the following classes: *simulation-based* and *formal representation-based* approaches. This chapter presents a survey of the existing works and it is structured in four sections: the first section presents the simulation-based works, the second section discusses the formal representation-based works and the third compares the work proposed in this thesis with the related work. The last section gives the conclusions.

1.1 Simulation – Based Works

The components that form a heterogeneous system are specific to different application domains. In a heterogeneous design environment the co-simulation requires significant test and modeling capabilities, not only for the specific technologies (continuous or discrete-only domains) but also for the technologies combination. There are two opinions regarding the co-simulation of heterogeneous systems: one that supports a homogeneous approach and the second one that supports a heterogeneous approach [5].

1.1.1 Homogeneous Approach

The homogeneous approach consists of the use of only one language for the global specification of the behavior of the system; hence the representation of different parts is realized in one simulation language. The language has to have a rich and consistent semantics in order to support the heterogeneity of a complex system. The main challenge of this approach is the difficulty to find such languages and this leads to the development of new languages and this is costly in terms of training time and development of new libraries time. One can observe two strategies (techniques):

- The extension of existing tools and languages. Most of the tools created using this approach started from classical HDLs (i.e. VHDL ([6]), Verilog ([7])) to which new concepts specific to other domains such as Analog Mixed Signal

(AMS) are added (i.e. the IEEE standards VHDL-AMS ([9], [10], [11]) and Verilog-AMS ([12], [13]).)

- The addition of an executable extension to a language that exists already (i.e SystemC ([14], [15]) that is an extension of C++).

VHDL-AMS is an extension of VHDL that can be used for modeling and validation of continuous/discrete systems. The modeling can be realized according to few categories of models: functional, behavioral, structural and physical. The VHDL scheduler was modified in order to take into account the analog solver. The developers added new objects and types and also new attributes for signals. VHDL-AMS is useful for the design of analog/digital systems but it is not powerful enough for higher levels of abstraction. [9] presents a behavioral model realized with VHDL-AMS. The authors add new concepts such as data types, analog and digital, functionality in continuous time, functionality controlled by events as well as analog-digital interactions.

Verilog-AMS ([13]) allows the designers to create and use modules that can encapsulate behavioral descriptions at high levels of abstraction as well as structural description of systems and components.

SystemC-AMS ([14]) is an extension of SystemC that was developed for continuous time systems modeling and simulation. Between other requirements, it has to provide a way to manage the interactions between the different models of computation and to support existing continuous time simulators. Therefore, the developers have to implement a library of components and solvers able to solve differential and algebraic equations. However, even if SystemC is a viable option for high level modeling and its AMS extension will improve its capabilities to provide a global co-simulation model for a continuous/discrete heterogeneous system, it is difficult to make it more powerful than the existing tools for analog simulation such as Matlab - Simulink® ([16]), mostly on the simulation precision level, availability of libraries. The examples provided in [15] are limited to the communication and signal treatment domains where the time advancement is realized with fix steps. However, this is not the case with other fields

like mechanical, electrical, micro-electro-mechanical systems (MEMS) or optical micro-systems where solvers with a variable step are required. This approach is interesting because it gives the possibility to use a synchronization mechanism for other systems' integration and a solver for complex systems and for levels of abstraction that are not normally covered by SystemC-AMS [15].

In [17] Patel and Shukla propose the extension of the modeling and simulation framework of SystemC by adding a number of cores specific for different models of computation: Syncronous Data Flow (SDF), Communicating Sequential Processes (CSP) and Finite State Machine (FSM). The simulation core of SystemC is implemented mostly for Discrete Event (DE) semantics. The cores proposed by [17] can be used with the SystemC discrete events core and it allows the developers to model and simulate specific heterogeneous systems such as SDF, CSP and FSM. The authors show with few examples that when using the specific cores, SystemC precision improved and simulation efficiency increased.

In all the tools presented in this section, the extensions are usually designed from scratch and by consequence their libraries are not as strong as the well established tools for the continuous field (i.e., Simulink®).

1.1.2 Heterogeneous Approach

The heterogeneous approach consists of the use of different languages that are specific for different sub-systems domains, therefore, they conserve the domain specific descriptions of the modules and the models are simulated in parallel. This task can be difficult because the simulation models are different and the global co-simulation requires a model that describes the synchronization and the interconnections between the sub-systems. The advantage of this approach is that each model can be described with a specific language and this allows for the exploitation of the best performances of the existing languages.

Some of the tools that use this approach are: Ptolemy developed by University of California at Berkeley ([18]), Chatoyant developed by University of Pittsburgh ([19], [20]) and the work realized at Techniques of Informatics and Microelectronics for integrated systems Architecture Laboratory (TIMA) in France ([5], [21], [22]).

Ptolemy ([18]) is a flexible design base that the developers can use to build prototyping environments. It supports heterogeneity and provides a tool to explore different design methodologies that support different types of design and implementation technologies. The models are built with different models of computation that characterize the behavior of the different parts of the system. Ptolemy II introduces the notion of *director* that encapsulates the behavior of a model of computation. Some of the supported directors are DE, SDF for the behavior of discrete events and synchronous data flow and CT - continuous time modeling. In terms of design, the models are implemented as an ensemble of components that communicate, named *actors*. The *actors* can communicate one with each other and they can execute simultaneously, the components are defined using an *actor oriented* approach. The communication is done via *channels* and the connection is through *actors*' ports. The only interaction between the *actors* is through their *channels*. Ptolemy II also supports the *hierarchical actors* notion where *actors* can contain other *actors* and that are connected by external ports.

The components can be developed to work with multiple data types. One of the types introduced here is the behavioral type. The components and the domains support interface definitions that describe not only the static structure like the traditional systems but also the dynamical behavior. HyVisual is a hybrid systems modeler built on top of Ptolemy II [23] that supports the construction of hierarchal hybrid systems for continuous-time dynamical systems and hybrid systems.

Even though Ptolemy II is an open source code and it is an extension of Java, it is a new language and using it requires learning time for the user. The different sub-systems and components need to be developed in the same environment in order to be compatible thus they do not solve the problem of IP reuse in system design. Moreover, Ptolemy is

based on formal representation, but the formal verification of the simulation models is not considered. It also lacks of consistency for analysis and verification during conception stage ([18], [24]). Moreover the execution angles (hardware) are not taken into consideration.

Chatoyant ([19], [20]) is a simulation environment that is an extension of Ptolemy environment based on an architecture design methodology at system level. The system is decomposed into component modules that are individually characterized. The information exchanged between the modules is determined by the components: optical, electrical and mechanical. The tool can realize static and dynamical end to end simulations. The static simulations analyze the mechanical tolerances, power loss and the dynamical simulations are executed to analyze the data flow with techniques like noise analysis.

One of Chatoyant applications is the modeling of optical interconnects. Its optical libraries include passive and active optical components, optical detectors and light sources. The optical signals are represented using “linear discrete events” techniques. In order to support micro-opto-electro-mechnical (MOEM) systems, Chatoyant was extended as follows:

- Introduction of modeling techniques for diffractive optics that allow the use of diffractive models in cases where the Gaussian approximations are not valid.
- Introduction of new models for micro-lenses, micro-mirrors and mechanical actuators that allow the global simulation of the system in one mixed signal frame.
- Implementation of a Monte Carlo tolerance package to determine the worst tolerance case and the mechanical stability.

The researchers from TIMA Laboratory defined a new model for the global representation of heterogeneous systems by automatic generation of co-simulation instances [21]. The heterogeneity is given by the co-existence of different modules described at different levels of abstraction, using, for the modules specification,

different languages. This concept makes possible the systems validation during different stages in the design flow. The methodology implies the generation of the simulation bus, the simulation interfaces as well as the communication interfaces at each level of abstraction [5]. The methodology also allows the description of each module in a language specific to its domain (i.e. SDL, VHDL, ISS) and at given level of abstraction.
The researchers from TIMA introduced the concept of *virtual architecture* that is a non executable model that represents the first step of the methodology of the automatic generation of the co-simulation models. One of the basic concepts proposed is the module's *wrap* that represents the abstraction of the interconnects between two heterogeneous components. Each *wrap* has a set of two *ports*: *internal ports* that are the module's ports and *external ports* that are the ports that allow the connection with the communication channels. A *module* and its *wrap* form a *virtual component*. The different communication channels connected to a virtual port can be grouped in *virtual channels*. Using these concepts the systems will be represented by a *virtual architecture* as a set of *virtual components* interconnected. For the automatic generation of co-simulation models at input of the design flow we have the description of the *virtual architecture* of a heterogeneous system and with elements from the co-simulation library, the co-simulation instances are generated. The strategy consists of the assembly of the existing elements into a co-simulation library. The main steps of the automatic generation of co-simulation models flow are:

- The first step consists in the analysis of the virtual architecture in order to collect the in formations necessary for the following steps.
- The second step consists in the selection of the library elements from the co-simulation library and the generation of the co-simulation interfaces. The selection is done using the results of the analysis of the system's specifications.
- The third step consists of the assembly of the system's components needed for a co-simulation instance. During this stage the co-simulation interfaces and the co-simulation bus are considered (introduced) in the initial structure of the system.

With this approach the verification/validation is realized by co-simulation. The static analysis is to check function coherence and to minimize the inter-functions coupling [5]. TIMA approach is used for hardware/software co-simulation and not for continuous/discrete models.

1.2 Formal Representation – Based Works

Formalism-based approaches model systems using a mathematical language like sets theory or systems theory or other formalized paradigm [4]. The integration is addressed as a composition of models of computation. These approaches propose a single main formalism to represent different models and the main concern is building interfaces between different Models of Computation (MoC). These approaches bring a deep conceptual understanding of each MoC.

The works that can be included in the formal representation – based approach were done at the University of California at Berkeley [25], the Royal Institute of Technology from Stockholm [26], [27] and the University of Arizona [28], [29] and briefly presented in this section.

In [25] a formal framework for comparing different models of computation used in heterogeneous models is presented. The authors propose a formal classification framework that makes possible to compare and express the differences between them. The framework was used to compare certain features of various MoCs such as dataflow, sequential processes and concurrent sequential processes with rendezvous, Petri nets, and discrete-event systems. The intent is “to be able to compare and contrast its notions of concurrency, communication, and time with those of other models of computation” [25].

The role of the model of computation in abstracting functionalities of complex heterogeneous systems was given in [27]. A study on the use of different models of computation for the formalization of complex heterogeneous systems functionalities is presented in [26]. The author proposes a formal framework by separating the

communication and the computation aspects. The process is divided into two parts: the *core* that is in charge with the computation and the process *shell* in charge with the communication with other processes. This separation gives a better comprehension of different problems. The designers do not have to take into consideration the problems raised by process computation while they are working on other subjects such as the communication the synchronization or the concurrence. Moreover, from a practical point of view, each part can be developed separately, integrated easier and also reused for other applications. All these elements are taken into consideration for the models of computation classification from a denotational point of view: untimed models of computation, timed models of computation, and synchronous models of computation. However, the interfaces between domains were not taken into consideration.

A meta-model named Rugby [26] that can be used for elements representation in terms of *domains*, *levels of abstraction* was defined. Rugby identifies four sub-domains: *computation, communication, domain* and *time*. The domains can be represented at different abstraction levels, from physical level to more abstract system levels (i.e. the time can be represented as continuous, discrete, clock and a causality relation).

DEVS (Discrete Event Systems Specifications), defined in [28], [29] is a mathematical formalism for systems representation and simulation where the time advances on a continuous time base. This approach is based on the systems theory: a system with a time base, inputs, states, outputs. Given the current states and the inputs, functions are implemented to determine the next states and the outputs. DEVS is a formal approach to build the models, using a hierarchical and modular approach and more recently it integrates object-oriented programming techniques. Based on this formalism, [30] has proposed a tool for the modeling and simulation of hybrid systems using Modelica and DEVS. The models are “created using Modelica standard notation and a translator converts them into DEVS models” [30]. In [31], the authors propose a heterogeneous simulation framework using DEVS BUS. Non-DEVS-compliant models are converted through a conversion protocol into DEVS-compliant models. CD++ [32] is a general

toolkit written in C++ that allows the definition of DEVS and Cell-DEVS models. DEVS coupled models and Cell-DEVS models can be defined using a high level specification language. PythonDEVS [33] is a tool for constructing DEVS models and generating Python code. A model is described by deriving coupled and/or atomic DEVS descriptive classes from this architecture, and arranging them in a hierarchical manner through composition. DEVSim++ [34] is an environment for Object-Oriented Modeling of Discrete Event Systems.

However, DEVS allows the definition of the operational semantics for a system but not its formal verification.

The rollback is also presented in several works. [35] proposes a rollback algorithm for optimistic distributed simulation systems. In [36] and [37] the authors detail different checkpoint mechanisms that allow the system's rollback in order to recover the data. [38] presents the "time warping" algorithm that allows the rollback to a point where data consistency is guaranteed. However, the formalization and verification of the rollback mechanism in the context of C/D was never addressed.

1.3 Research Project vs. Related Work

Compared with the existing works, this thesis combines the two approaches: simulation-based and formal representation-based validation. We define here a generic methodology for the efficient design of continuous/discrete co-simulation tools that will improve upon some of the deficiencies observed in the works prior presented.

The homogenous simulation–based approaches imply the development of new languages that are costly in terms of training time and development of new libraries. In the case of the heterogeneous simulation-based approach the interfaces are developed ad-hoc and they are not formally verified. Moreover the developers do not focus on the continuous/discrete interfaces. This thesis introduces an approach where the developers can use for each domain existing powerful tools and reuse models that were already

tested in the simulation context. The interfaces are formally verified and automatically generated.

The formal representation-based approaches provide an abstract base for heterogeneous systems' representation but they do not take into consideration the co-simulation interfaces or they do not allow for the formal verification. In our work, we focus on the co-simulation interfaces and we provide a mechanism for the formal representation and formal verification of the co-simulation interfaces.

The advantage of this methodology is the convergence of the formal representation and the co-simulation in the context of global validation of continuous/discrete systems. We combine here the rapidity of the co-simulation technique with the completeness of the formal verification.

The methodology includes the definition of the operational semantics for continuous/discrete synchronization models as well as the formal representation and verification of the behavior of continuous/discrete co-simulation interfaces and their internal architecture. Moreover, it allows the representation of the continuous and the discrete in well established languages and by consequence the use of the libraries that are already tested and used. The users do not need to learn new languages and can reuse IPs in system design.

1.4 Conclusion

This chapter presented a survey of the existing works proposed for the heterogeneous systems validation. There are roughly two strategies that are widely accepted: simulation-based and formal-based representation approaches. Some of the simulation-based validation tools use a single language for the specification continuous/discrete system (homogenous simulation-based validation). These tools may be obtained by extension of existing HDLs (VHDL-AMS, Verilog-AMS and SystemC-AMS). Other simulation-based validation tools assemble together different components in order to

generate the global system (heterogeneous simulation-based validation). Some of these tools are Ptolemy, Chatoyant and the model developed by TIMA Laboratory

The formal representation-based approaches propose different definitions for heterogeneous systems modeling. The works briefly presented in this chapter were realized at the University of California at Berkeley [23], the Royal Institute of Technology from Stockholm [26], [27] and the University of Arizona [28], [29].

This thesis introduces a new perspective: it unifies these two approaches. The result is a new generic methodology for the design of efficient continuous/discrete co-simulation tools that has the advantages of both techniques mentioned above.

CHAPTER 2. EXECUTION AND SYNCHRONIZATION MODELS

This chapter presents the global execution models of continuous/discrete heterogeneous systems. The chapter is organized in four sections. Section 1 defines the global execution model for a continuous/discrete heterogeneous system. Section introduces the synchronization models: the canonical model and the rollback-based model. Section 3 presents the events update schemas for the discrete simulator and Section 4 gives the conclusion.

2.1 Global Execution Model

Figure 2.1(a) shows a generic C/D system and Figure 2.1(b) shows its corresponding global execution model.

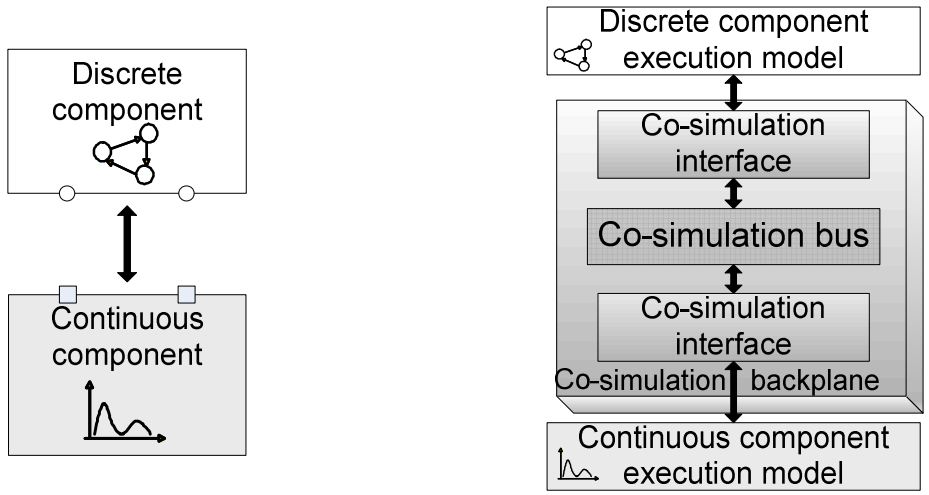


Figure 2.1. A continuous/discrete global execution model

There are three types of basic elements that compose this model [21] :

- The *execution models* of the different components constituting the heterogeneous system (corresponding to Continuous component and Discrete component in Figure 2.1).

- The *co-simulation bus*.
- The *co-simulation interfaces*.

The co-*simulation bus* is in charge of interpreting the interconnections between the different components of the system.

The co-*simulation interfaces* enable the communication of different components through the simulation bus. They are in charge of the adaptation of different simulators to the co-simulation bus in order to guarantee the transmission of information between simulators executing the different components of the heterogeneous systems. They also have to provide efficient synchronization models for the modules adaptation.

The co-*simulation backplane* is the element of the global execution model that guarantees the synchronization and the communication between the different components of the system. It is composed of the above mentioned simulation interfaces and the simulation bus.

The implementation and the simulation of an execution model in a given context is called co-*simulation instance*. Several instances may correspond to the same execution model and these instances may use different simulators and may present different characteristics (e.g. accuracy and performances).

2.1.1 Discrete Execution Model

The execution model for a discrete system is a model where changes in the state of the system occur at discrete points in the execution time.

The discrete system can be described by the state-space equations [38]:

$$\begin{cases} x_d(t_{k+1}) = f(x_d(t_k), u(t_k), t_k) \text{ with } x(t_0) = x_0 \\ y(t_k) = g(x_d(t_k), u(t_k), t_k) \end{cases} \quad (1)$$

Where f and g are transformations, x_d is the discrete state vector, u the input signal vector, and y the output signal vector.

A system modeled through (1) is said to be linear if and only if the functions g(·)and f (·) are both linear ([38]).

For the linear discrete systems, (1) becomes:

$$\begin{cases} x_d(t_{k+1}) = A_d x_d(t_k) + B_d u(t_k) \\ y(t_k) = C_d x_d(t_k) + D_d u(t_k) \end{cases} . \quad (2)$$

where A_d, B_d, C_d and D_d are matrixes that can be time-varying and describe the dynamics of the system.

If we conside n state variables, m output variables, and p input variables, then $A_d(t_k)$ is an $n \times n$ matrix, $B_d(t_k)$ is a $n \times p$ matrix, $C_d(t_k)$ is a $m \times n$ matrix, and $D_d(t_k)$ is a $m \times p$ matrix. The class of linear systems is a small subset of all possible systems but it covers many cases of interest, or provides adequate approximations can be used [38].

A discrete-event system execution concentrates on processing events, each event having assigned a time stamp. Each event computation can *modify* the state variables, *schedule new events* or *retract* existing events. The unprocessed events are stored in a pending events list. The events are processed in the order of their time stamp. Figure 2.2 shows a possible update event schema.

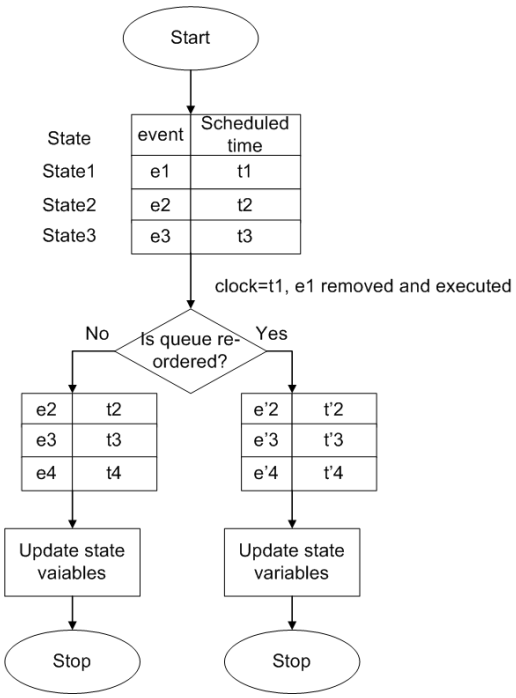


Figure 2. 2. Event update schema in a discrete simulator ([29])

At each simulation cycle, the first event with the smallest time stamp is processed and the processes sensitive to this event are executed. If several processes are sensitive to one or several events (with the same time occurrence) then these processes have to be executed in parallel. Executions often occur on sequential machines that can only execute one instruction at a time (therefore, one process). The consequence is that this execution cannot parallelize the processes. The solution consists in emulating the parallelism, where the processes are executed as if the parallelism is real and the environment (its inputs) does not change while executing other processes. Thus, the process execution order loses its importance and everything takes place as if a parallel execution occurred. This requires that shared variables (signals) between processes keep their values until the execution of all parallel processes ends. Once all events with discrete time stamp equal to the current time have been treated, the simulator advances the time to the nearest scheduled discrete event.

Illustrative examples of discrete-time simulators are: SystemC [8], VHDL [6], Verilog [7], SystemVerilog [39].

2.1.2 Continuous Execution Model

The continuous time execution model is described by the state space equations:

$$\begin{cases} \dot{x}_c(t) = A_c x_c(t) + B_c u(t) \\ y(t) = C_c x_c(t) + D_c u(t) \end{cases}. \tag{3}$$

where x_c is the state vector, u the input signal vector, y the output signal vector and A_c, B_c, C_c and D_c are constant matrixes that describe the dynamic of the system. The execution of continuous model, described by differential and algebraic equations, requires solving numerically these equations. A widely used class of algorithms [40] discretizes the continuous time line into an increasing set of discrete time instants, and computes numerically values of state variables at these ordered time instants. The interval between two consecutive time instants is called integration step, and can be

fixed or variable. The criteria used for the choice of the integration step are the accuracy, the stability and the continuity of the signals. The next state of derivative systems cannot be specified directly but the derivative functions are used to specify the rate of change of state variables [29].

The execution of a continuous system raises problems because given a state q_k and a vector x for a time t_k, the derivative offers information only for dq_k/dt but not the system's behavior over time. For a nonzero interval $[t_k, t_{k+1}]$ the computation has to be realized without knowing the behavior in the interval (t_k, t_{k+1}). This problem can be solved using numerical integration methods. Some of the most commonly used methods are [29]:

- *Euler method* that consists in signal integration:

  $$\frac{dq(t)}{dt} = \lim_{h \to \infty} \frac{q(t+h) - q(t)}{h}$$

  For an h small enough (in order to obtain accurate results), the following approximation can be used:

  $$q(t+h) = q(t) + h * \frac{dq(t)}{d(t)}$$

  This solution has low efficiency and does not have stability problems for small enough h and it is very robust.
- *Causal methods* that are a linear combination of states and derivative values at time instants with coefficients chosen to minimize errors from the computed estimate to the real value. This solution has high efficiency but it has stability and robustness problems.
- *Noncausal methods* that use "future" values of states, derivative and inputs. In order to do that, the model is executed past the needed time and the values that are necessary are stored, to estimate the present values.

Table 2.1 shows the difference between the basic concepts for the continuous and the discrete models.

Table 2.1. Continuous system vs. discrete system

Model\Concept	Time	Communication means	Processes activation rules
Discrete	It advances discretely	Set of events	Processes are sensitive to events
Continuous	It advances by integration steps	Piecewise continuous signals	Processes are executed at each integration step

The concepts taken into consideration here are the time, the communication means and the processes activation rules.

Illustrative examples of continuous-time simulators are: Simulink® [16] and SPICE [41].

2.2 Continuous/Discrete Synchronization Models

This section proposes two synchronization models for the global execution of C/D heterogeneous systems:

- the *canonical model* where the continuous simulator advances before the discrete simulator.
- the *rollback-based model* where the discrete simulator advances before the continuous simulator.

For these models we consider $[t_k,t_{k+1}]$ as the time interval. The input signal vector for the continuous domain is the output signal vector from the discrete domain and vice versa. The simulation of discrete models is based on *events* [42]. At each simulation cycle, the first event with the smallest time stamp is processed and the processes sensitive to this event are executed. This may generate other events causing execution of other processes. Once all events with discrete time stamp equal to the current time have been treated, the simulator advances the time to the nearest discrete scheduled event.

The events exchanged between the discrete and the continuous simulators are [42]:

- *discrete events* are timed events scheduled by the discrete simulator. The events sent by the discrete simulator can be signals update events that are caused by the change of its input discrete signals or sampling events that are pure events (defined only by their time stamps) and indicate the sampling events time stamps.
- *state events* are unpredictable events generated by the continuous simulator. Their time stamp depends on the values of state variables (e.g. a zero-passing or a threshold crossing).

When stepping ahead in time, a simulator must consider the events time stamps coming from the external world and it must reach accurately these time stamps of events (called here *events detection*). These time stamps are the synchronization and communication points between the different simulators involved in a global simulation. For a rigorous synchronization each simulator has to detect, locate in time and react to events sent by the other simulator.

2.2.1 Continuous/Discrete Canonical Synchronization Model

This sub-section details the C/D canonical synchronization model. In order to avoid the discrete simulator backtracking we have to detect the *state events* generated by the continuous simulator before the advance of the discrete simulator time, therefore the continuous simulator has to advance before the discrete simulator [43].

Figure 2.3 presents the synchronization model in the continuous/discrete co-simulation interfaces without state event (Figure 2.3(a)) and with state event (Figure 2.3(b)). At a given time the discrete simulator is in the state s_{dk} that is the tuple (x_{dk},t_k) where x_{dk} is the location and t_k is the k^{-th} discrete time (that can be seen also as the k^{-th} event in the queue of events in the discrete domain). At this point the discrete simulator had executed all the processes sensitive to the event and sends the time of the next event t_{k+1} and the data to the continuous simulator and switches the context from the discrete to the continuous simulator before advancing the time (arrow 1 in Figure 2.3(a) and Figure 2.3(b)).

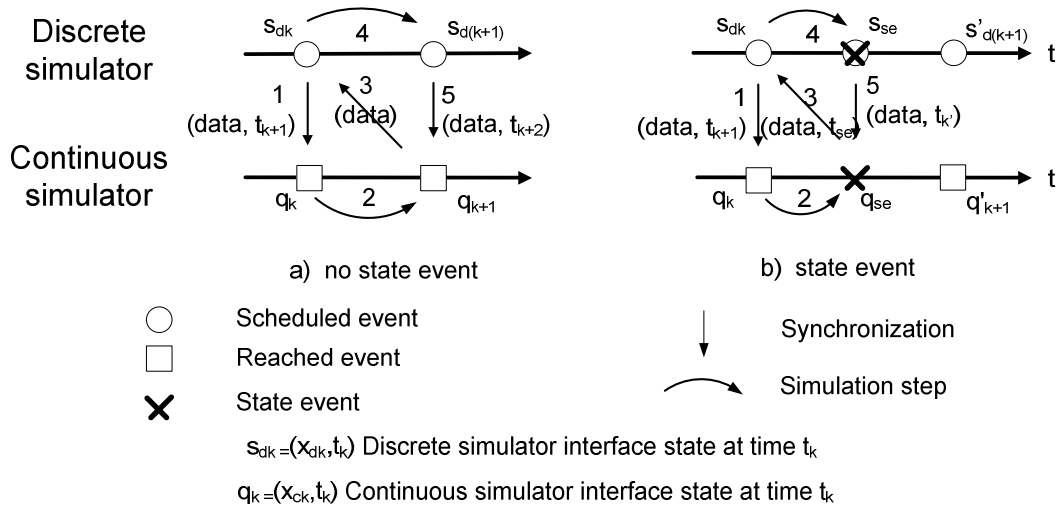


Figure 2.3. The canonical synchronization model

The state of the continuous simulator is q_k that is the tuple (x_{ck},t_k) and the advance in time of the simulator cannot be further than t_{k+1}, the time sent by the discrete simulator. The behavior of the continuous interface can be described by the following transition state equation (arrow 2 in Figure 2.3(a) and Figure 2.3(b)):

$$(x_{ck},t_k) \rightarrow \begin{cases} (x_{ck+1},t_{k+1}) \; if \; t = t_{k+1} & (4) \\ (se,t_{se}) \; if \; t < t_{k+1} & (5) \end{cases}.$$

where t is the time in the continuous domain, (x_{ck+1}, t_{k+1}) is the state of the continuous simulator when no state event was generated in the time interval $[t_k, ,t_{k+1}]$. The state q_{se} that is the tuple (se,t_{se}) represents the state of the continuous simulator when a state event se was generated and t_{se} represents the time when the state event occurred. In both situations the continuous simulator will stop and send the data to the discrete simulator and then switch the context to the time t_k (arrow 3 in Figure 2.3(a) and Figure 2.3(b)). The event taken into consideration is the event generated within the time interval $[t_k,t_{k+1}]$, after the context switch from the discrete domain to the continuous domain at the time t_k. This event can be a state event or the detection of an event scheduled by the discrete simulator (and consequently a synchronization point).

In the case described by equation (4), after switching the context, the discrete simulator will advance to the time t_{k+1} that is the next synchronization point, where it will execute all the processes sensitive to this event. Before switching the context to the continuous interface the discrete simulator sends the data and the time of the next scheduled event t_{k+2} (also the next synchronization point) and the cycle restarts (arrow 4 in Figure 2.3(a)).

Equation (5) describes the case where a state event occurred. The continuous simulator will send not only the data but also the time when the state event occurred t_{se} (arrow 3 in Figure 2.3(b)). The discrete simulator will advance to this time (state event detected by the discrete simulator) where it will execute all the processes sensitive to the event. Before switching the context to the continuous simulator the discrete interface will send the data and the recalculated time of the next scheduled event t_k (arrow 4 in Figure 2.3(b)). The time stamp can change after a state event. This time stamp can take any value bigger than t_{se} .The advantage of this model is that it avoids any need of rollback even if a state event was generated.

2.2.2 Continuous/discrete rollback-based synchronization model

Figure 2.4 presents the light rollback synchronization model for the C/D simulation interfaces.

At a given time the discrete simulator is in the state s_{dk} that is the tuple (x_{dk},t_k) where x_{dk} the location and t_k the k^{-th} discrete time (that can be seen also as the k^{-th} event in the queue of events in the discrete domain). At this point the discrete simulator had executed all the processes sensitive to the event, advances to the time of the next event t_{k+1} (arrow 1 in Figure 2.4(a) and Figure 2.4(b)) and a new state s_{dk+1} that is the tuple (x_{dk+1},t_{k+1}), sends the data and the time of the event t_{k+1} to the continuous simulator and switches the context to the continuous simulator (arrow 2 in Figure 2.4(a) and Figure 2.4(b)).

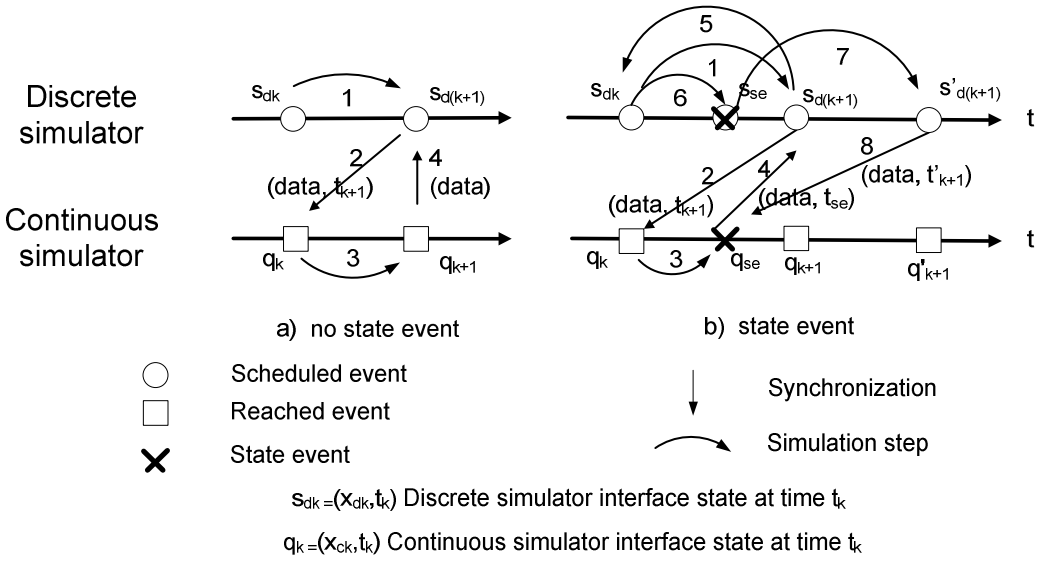


Figure 2.4. The rollback-based synchronization model

The state of the continuous simulator is q_k that is (x_{ck},t_k) and the advance in time of the simulator cannot be further then t_{k+1}, the time sent by the discrete simulator.

The behavior of the continuous interface can be described by the same transition state equation that was presented for the canonical synchronization model.

The case described by equation (4) is the case when the continuous simulator does not send state events. In this case the continuous simulator will behave like in the case of the canonical synchronization model (the equation (4) was already presented in section 2.2.1) and is represented in Figure 2.4(a). Equation (6) describes the case where a state event occurred. In the case of the rollback-based synchronization model the continuous simulator will send not only the data but also the time when the state event occurred t_{se} (arrow 4 in Figure 2.4(b)). The discrete simulator will backtrack to the previous state s_{dk} (arrow 5 in Figure 2.4(b)) and restores the saved data for the time stamp t_k. This backtrack where only a backup of memory data segment, processor registers as well as input and output signal values will be made for each discrete event is called here light rollback. After the initial state restoration, the simulator starts over, taking into account the state events and advances to the time stamp t_{se} (state event detected by the discrete

simulator) where will execute all the processes sensitive to the event (arrow 6 in Figure 2.4(b)). The cycle restarts, the discrete time advances to the next discrete event. The time stamp of this event can change after a state event; it can take any value bigger than t_{se} .

2.3 Events Update Schema

In both synchronization models a key point is represented by the events update schema in the discrete domain. This section presents step by step these schemas for a discrete simulator integrated in a continuous/discrete co-simulation environment. The elements used in this representation respect the definitions introduced in [38]:

- The system maintains a Scheduled Event List $L=\{(x_{dk},t_k)\}$ with k=1,2,3,...n. The list is ordered on the smallest-first basis.
- The queue of events is ordered by the events lifetimes, from the smallest to the largest. The lifetime v_k is the length of the time interval between two successive occurrences of an event ($v_k=t_{k+1}-t_k$).

Considering that the list is reordered each time the context is switched from the continuous domain to the discrete domain, some events will become undetectable so they have to be deleted from the list or new events will be generated and therefore they have to be added to the list. There are two possible behaviors of the scheduler, both of them depending on the behavior of the continuous domain:

- When no state event occurred in the continuous domain;
- When a state event was generated in the continuous domain.

In both cases `State` is initialized to a given value x_0 and the simulation time `Time` is initialized to 0. The *Clock Structure* is a set of clock sequences, one for each event.

The next two sub-sections present the events update schema for both models of synchronization: the canonical model and the rollback-based model.

2.3.1 The Event Update Schema for the Canonical Discrete Simulator

Figure 2.5 presents the event update schema for a canonical discrete simulator integrated in a continuous/discrete co-simulation environment. This figure is inspired by [38]. In [38] the author proposed the event update schema for a purely discrete event system. Figure 2.5 extends this schema with the interaction in terms of communication/synchronization (through the events exchanged) between the discrete and the continuous simulators.

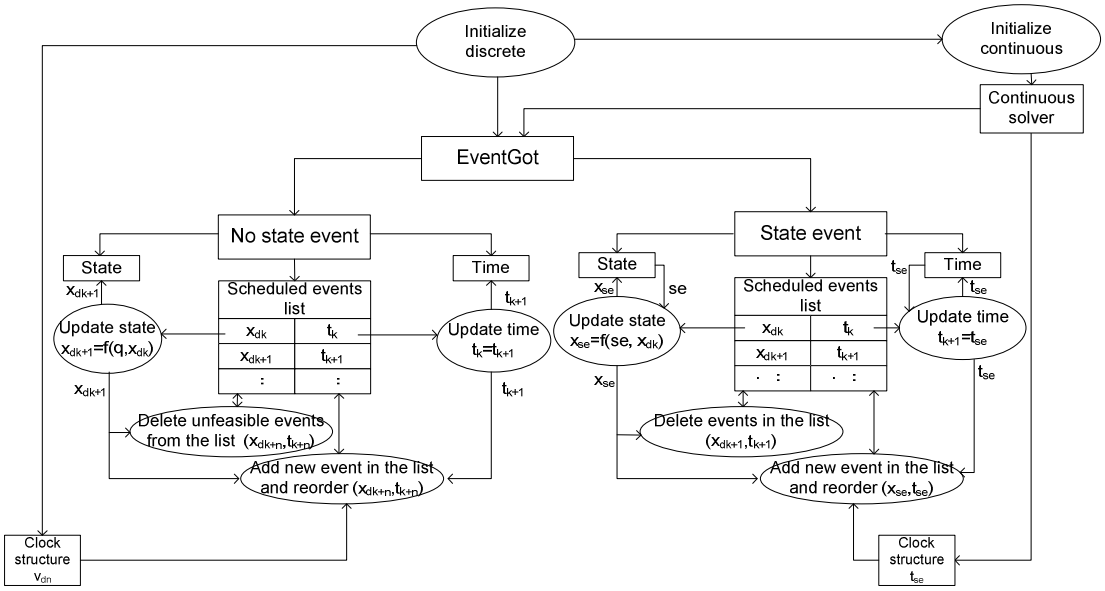


Figure 2.5. The event update schema for the canonical discrete simulator

For the case when no state event is generated the following steps are executed (see Figure 2.5):

Step1 - First entry in the list (x_{dk},t_k) is removed from the list.

Step2 - Time is updated to a new time.

Step3 - State is updated according with the transition function, $x_{dk+1}=f(q,x_{dk})$ where q is the data from the continuous domain.

Step4 - The events that became unfeasible after the data is received from the continuous domain are deleted from the list.

Step5 - New feasible events that are a consequence of the data received from the continuous domain are added to the list.

Step6 - The list is reordered on the smallest-first basis.

The procedure repeats with step 1 for the new list. In this case the clock structure is controlled by the discrete domain; the events queue is reordered by the discrete kernel.

When the continuous domain generates a state event the sequence of steps is the following (Figure 2.5):

Step1 - First entry in the list (x_{dk},t_k) is removed from the list.

Step2 - Time is updated to a new time $t_{se} < t_{k+1.}$

Step3 - State is updated according with the transition function, $x_{se}=f(se,x_{dk})$ with q the data from the continuous domain.

Step4 - The state event is added in the list always as the next entry to be removed from the list.

Sep5 - The events that became unfeasible as a consequence of the detection of a state event (which in an unpredictable event) are deleted from the list.

Step6 - New feasible events that are a consequence of the state event are added to the list

Step7 - The list is reordered on the smallest-first basis.

This procedure repeats with step 1 for the new list. In this case the clock structure is controlled by the continuous solver, the time of the state event is sent by the continuous domain and the first consequence is the re-start of the discrete simulator at a time t_{se}, before the expected time t_{k+1}.

2.3.2 The event update schema for the rollback-based discrete simulator

Figure 2.6 presents the event update schema for a rollback-based discrete simulator integrated in a continuous/discrete co-simulation environment. For the first case when no state event is generated the following steps are executed (see Figure 2.6):

Step1 - First entry in the list (x_k,t_k) is removed from the list.

Step2 - Time is updated to a new time.

Step3 - State is updated according with the transition function, $x_{dk+1}=f(q_k,x_{dk})$ with q_k the data from the continuous domain (a particular case of this step is the initial transition function when from (x_0,t_0) to (x_1,t_1) where $x_1=f(x_0)$).

Step4 - The events that became unfeasible after the data is received from the continuous domain are deleted from the list.

Step5 - New feasible events that are a consequence of the data received from the continuous domain are added to the list.

Step6 - The list is reordered on the smallest-first basis.

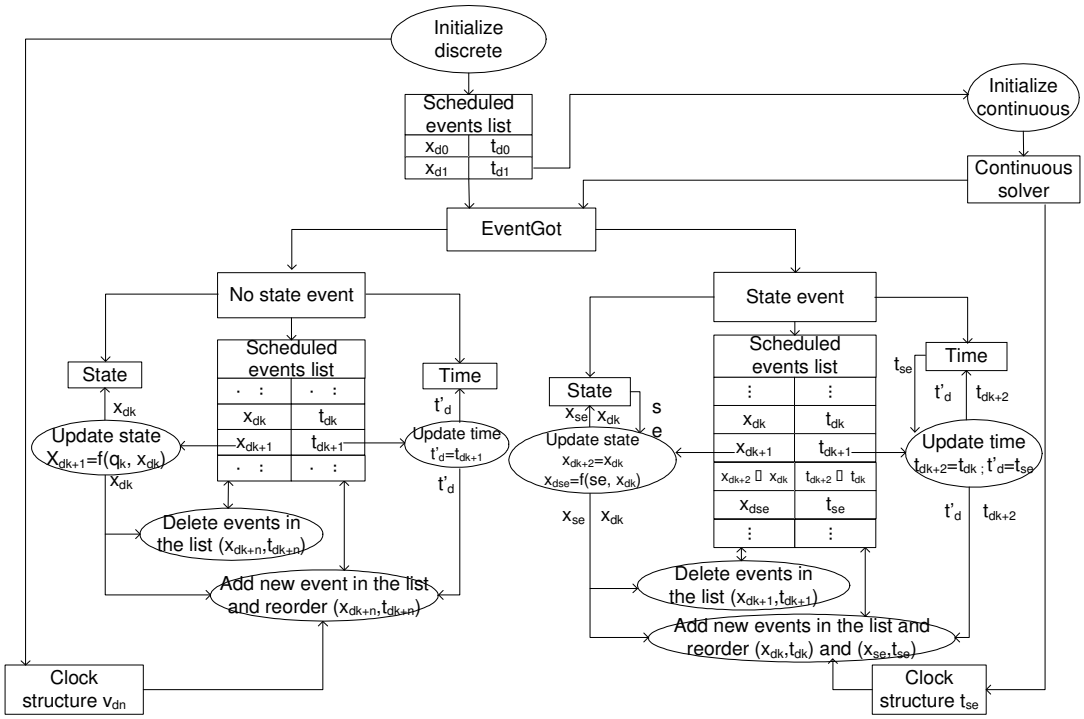


Figure 2.6 The event update schema for the rollback-based discrete simulator

The procedure repeats with step 1 for the new list. In this case the clock structure is controlled by the discrete domain; the events queue is reordered by the discrete kernel.

When the continuous domain generates a state event the sequence of steps is the following:

Step1 - First entry in the list (x_k,t_k) is removed from the list

Step2 – Time is updated to a new time.

Step3 - The second entry in the list (x_{k+1},t_{k+1}) is removed from the list.

Step4 - Time is updated to a new time.

Step5 - State is updated according with the transition function, $x_{se}=f(se,x_{dk})$ where q is the data from the continuous domain.

Step6 – First entry in the list (x_k,t_k) is added back to the list (data is recovered).

Step7 – Time is updates to a new time.

Step8 - The state event is added in the list always as the next entry to be removed from the list.

Step9 - The events that became unfeasible as a consequence of the detection of a state event (which in an unpredictable event) are deleted from the list.

Step10 - New feasible events that are a consequence of the state event are added to the list.

Step11 - The list is reordered on the smallest-first basis.

This procedure repeats with step 1 for the new list. In this case the clock structure is controlled by the continuous solver, the time of the state event is sent by the continuous domain and the first consequence is the re-start of the discrete simulator at a time t_{se}, before the expected time x_{tk+1}.

The synchronization models in C/D heterogeneous systems are presented in Table 2.2. We also give here the advantages and the disadvantages for each of the presented models.

Table 2.2. Synchronization in continuous/discrete heterogeneous systems

Synchronization model	Synchronization step	Advantages	Disadvantages
Canonical synchronization model	At each discrete step and state event occurrence	Can be applied for all systems	Synchronization overhead
Rollback-based synchronization model	At each update and sampling events and state event occurrence	Non-periodic update/sample events, it is efficient when no state events occurs	Rollback for discrete model is required if the continuous model generates state events

2.4 Conclusion

This chapter presented the global execution model of continuous/discrete heterogeneous systems. The first section introduced the main components of the global execution models: the domain specific execution models, the co-simulation bus and the co-simulation interfaces. The co-simulation interfaces have to provide efficient synchronization models. The second section of this chapter details two synchronization models: the canonical model and the rollback-based model. In the case of the canonical model the continuous domain simulator advances before the discrete domain simulator. The need for rollback is completely eliminated. In the case of the rollback-based model the discrete simulator advances before the continuous simulator and, if a state event is generated by the continuous domain, the discrete model will backtrack to the previous stable state.

This last section of the chapter presented the events update schema for the discrete simulator for both synchronization models.

CHAPTER 3. GENERIC METHODOLOGY FOR THE DESIGN OF CO-SIMULATION TOOLS

This chapter proposes a new methodology for the design of continuous/discrete co-simulation tools (as shown in Figure 3.1) divided in two stages: a generic stage and an implementation stage. This methodology presents several steps that are independent of the simulation tools used for the continuous and discrete components of the system. During these generic steps, the co-simulation interfaces are defined in a conceptual framework; their functionality and the internal structure of simulation interfaces are expressed using existing formalisms and temporal logic. After the rigorous definition of the required functionality for simulation interfaces, the designer will start the steps related to the implementation.

The main stages of the proposed methodology (illustrated in Figure 3.1) are:

1. A generic stage including the following steps:
    - Definition of the operational semantics for the synchronization in continuous/discrete global execution models.
    - Distribution of the synchronization functionality to the simulation interfaces.
    - Formalization and verification of the simulation interfaces behavior.
    - Definition of the library elements and the internal architecture of the simulation interfaces.
2. An implementation stage including the following steps:
    - The analysis of the simulation tools for the integration in the co-simulation framework.
    - The implementation of the library elements specific to different simulation tools and the implementation validation.

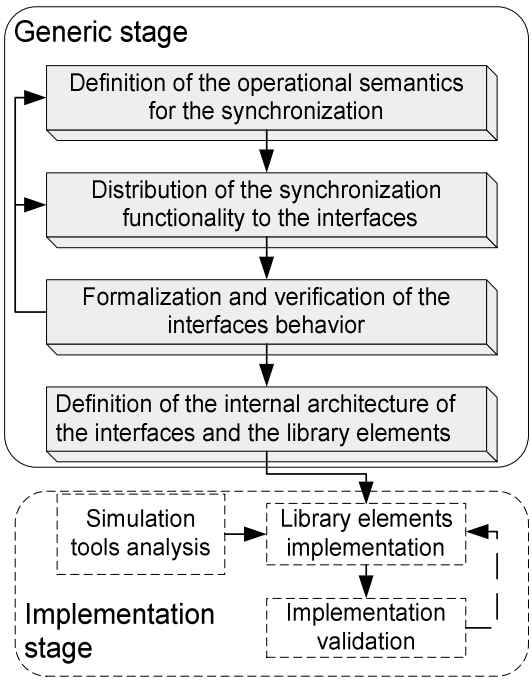


Figure 3.1. A generic methodology for co-simulation tools design

These steps will be detailed in the sub-sections of this chapter.

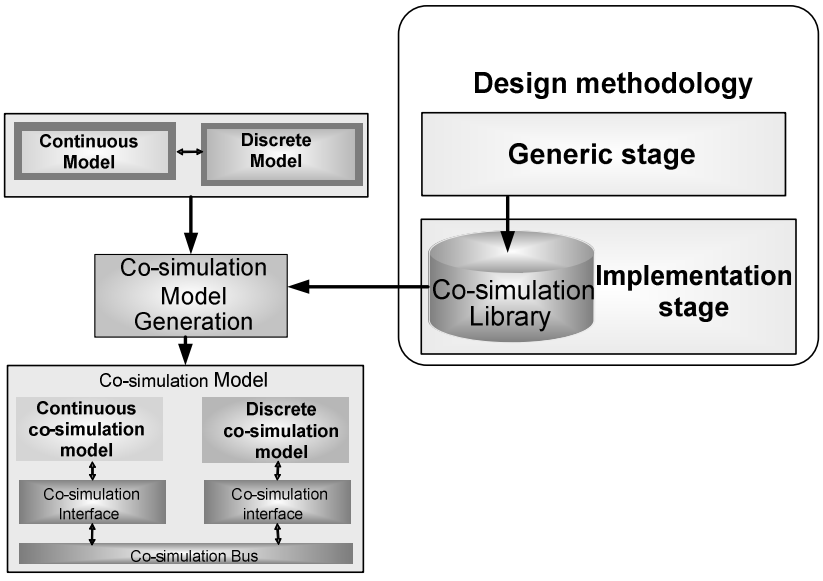


Figure 3.2. Design methodology in the flow for the automatic generation of co-simulation models

Figure 3.2 presents the proposed design methodology in the context of the automatic generation of execution models.

We emphasize here that the methodology is generic; the first stage is independent of the implementation languages of the co-simulation library.

Sub-section 3.1 “Generic Methodology” of this chapter generally presents the proposed methodology. Sub-section 3.2 “Using formal methods for co-simulation tools design” gives details on how this methodology can be applied, using existing formalism and tools.

3.1 Generic Methodology

This section focuses on the generic methodology and its stages. Each of the following sub-sections will detail these steps.

3.1.1 Definition of the Operational Semantics for the Synchronization in Continuous/Discrete Global Execution Models

The first step of the methodology for co-simulation tools design is the definition of the operational semantics for the synchronization in continuous/discrete global execution models. An operational semantics gives a detailed description of the system’s behavior in mathematical terms. This model serves as a basis for analysis and verification. The description provides a clear language independent model that can serve as a reference for different implementations.

The operational semantics for continuous/discrete systems requires the rigorous representation of the relation between the simulators (communication/synchronization and data exchanged between the continuous and the discrete simulators) as well as their high level and dynamic representations.

Figure 3.3 shows a view of the continuous/discrete heterogeneous during the “definition of the operational semantics for the synchronization” stage.

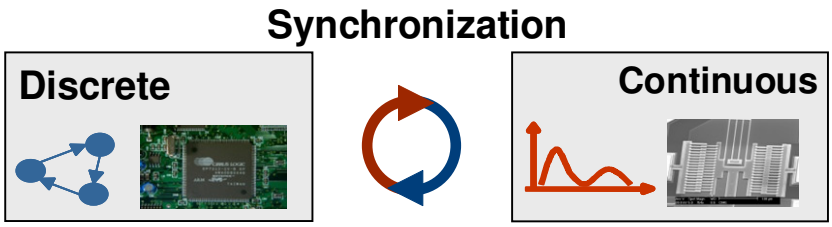


Figure 3.3. The continuous/discrete system during the "Definition of the operational semantics" stage

3.1.2 Distribution of the Synchronization Functionality to the Co-Simulation Interfaces

Based on the operational semantics, we can now define the synchronization functionality between the continuous and the discrete simulators. This functionality is insured by the interfaces that are the link between the different execution models and the co-simulation bus (see Figure 2.1). They are each in charge with a part of the synchronization between the two models. To insure system's flexibility, the synchronization functionality has to be distributed to the simulation interfaces. Moreover, each computation step has to be thoroughly specified. Figure 3.4 shows a view of the continuous/discrete heterogeneous during the "distribution of the synchronization functionality to the co-simulation interfaces" stage.

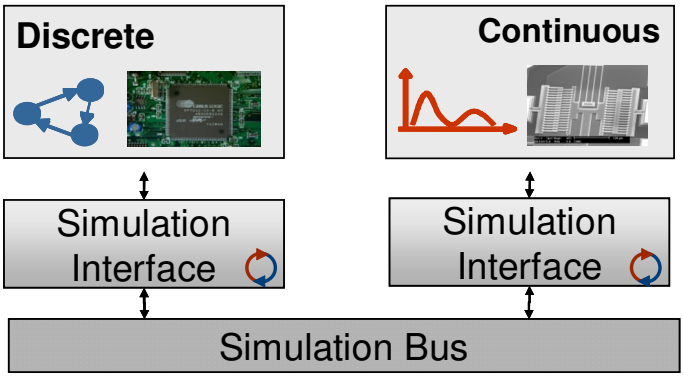


Figure 3.4. The continuous/discrete system during the "Distribution of the synchronization functionality to the co-simulation interfaces" stage

3.1.3 Formalization and Verification of the Simulation Interfaces Behavior

The formalization and verification of the simulation interfaces behavior stage can be roughly divided into three steps: formalization (that can be the formal specification of the heterogeneous system), the validation by model simulation and the formal verification. The two main techniques that can be used for the formal verification of the interfaces are [44] :

- *model checking* where the system descriptions are given as automata, the specification formulas are given as temporal logic formulas and the checking consists of the verification that all models of a given system description satisfy a given specification formula. It focuses mainly on automatic verification. Completeness and termination guarantee of model checking are some features of this technique, as well as it enables the tool to guarantee the correctness of a given property, or produce a counterexample otherwise.
- *theorem proving* where the verification plan is manually designed and the correctness of the steps in the plan is verified using theorem provers. Completely automatic decision procedures are impossible because the input language (the model and the specification) is of higher order logic and that eliminates the decidability. Moreover, everything has to be translated in higher order logic, and, therefore, the structure of the system may be lost and its representation can become large and difficult to work with.

Considering that the system is dynamic, it is necessary to use a formalism that allows the expression of dynamic properties (the state of a system changes and by consequence the properties of the state also change). The temporal logic handles formalization where the properties evolve over time and in general uses:

- propositions that describe the states (i.e., elementary formulas and logical connectors), and

- temporal operators that allow the expression of the properties of the states successions (called executions).

The differences between the logics are in terms of temporal operators and objects on which they are interpreted (such as sequences or state trees) [45].

The most commonly used logics are Linear Temporal Logic (LTL), Computation Tree Logic (CTL* and CTL, both of them untimed temporal logics) and their timed extensions TCTL and Metric Interval Temporal Logic (MITL).

- CTL* allows the use of all temporal and branching operators but the property verification is very complex. For this reason, most of the tools actually used allow the verification of fragments of CTL*.
- LTL is a fragment of CTL* that excludes the trajectory quantifiers. In this case only the trajectory predicates are considered. LTL does not provide a means for considering the existence of different possible behaviors starting from a given state (sequential) [45].
- CTL is also a fragment of CTL* and it is obtained when every occurrence of a temporal operator is immediately preceded by a branching operator. In the case of CTL we have state trees.
- TCTL is a timed temporal logic that is an extension of CTL obtained by subscribing the modalities with time intervals specifying time restrictions on formulas.

For our formal model, the properties that need to be checked are branching properties that are expressed using CTL or TCTL logics.

3.1.4 Definition of the Internal Architecture of the Simulation Interfaces

The formalization of the simulation interfaces behavior step is naturally followed by the definition of their internal architecture. This definition eases the automatic generation of

the simulation interfaces. We present in Figure 3.5 the hierarchical representation of the global simulation model used in our approach.

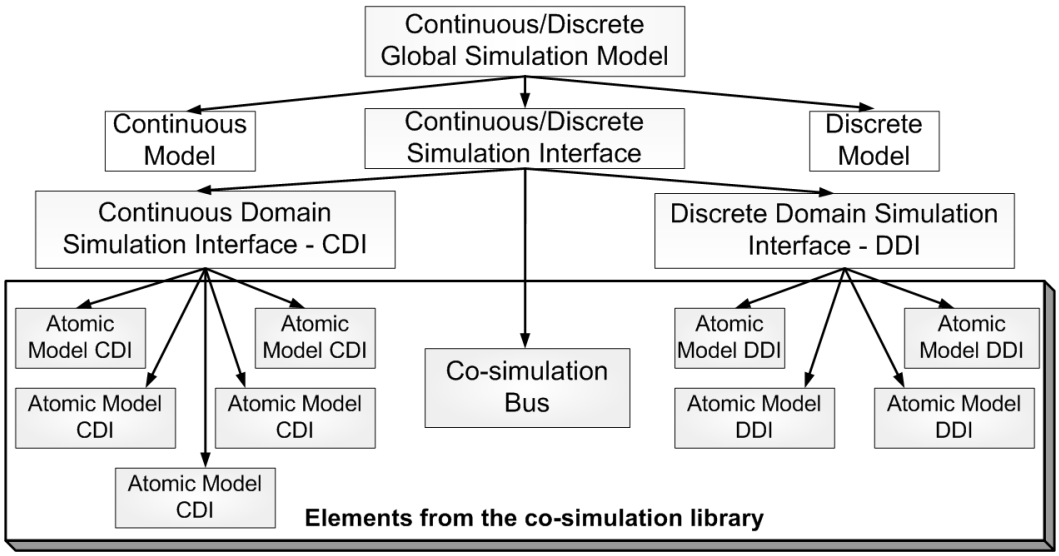


Figure 3.5. Hierarchical representation of the generic architecture of the co-simulation model

At the top hierarchical level, the global model is composed of the continuous and discrete models and of the C/D simulation interface required for the global simulation.

The second hierarchical level of the global simulation model includes the domain specific simulation interfaces and the co-simulation bus in charge of the data transfer between these interfaces.

The bottom hierarchical level includes the elements from the co-simulation library that are the atomic modules of the domain specific simulation interface. These atomic components implement basic functionalities of the synchronization model.

3.1.5 The Analysis of the Simulation Tools for the Integration in the Co-Simulation Framework

The considerations presented in the previous steps of the methodology show that specific functionalities are required for the co-simulation of continuous and discrete

models. Therefore, the integration of a simulation tool in the co-simulation environment requires their analysis. Thus, in the case of continuous simulator integration in the co-simulation tool, this simulator has to provide APIs enabling the following controls:

- State event detection and location.
- Setting break points during differential equation solving.
- On-line update of the breakpoints settings.
- Sending processing results and information for synchronization (i.e., the time step of the state event) to the discrete simulator. This implies generally the possibility to integrate C-code and Inter-Process Communications (IPC).

For the integration of a discrete simulator in the co-simulation tool, the simulator has to allow the addition of the following functionalities:

- Detection of the end of the discrete simulation cycle that guarantees that the simulation control is transferred to the continuous simulator only after the stabilization of discrete simulator.
- Insertion and retraction of new events (*state events*) in the scheduler's queue. This must be done before the advancement of the simulator time.
- Sending processing results and information for synchronization to the continuous simulator (i.e., the *time stamp* of its next discrete event).

3.1.6 The Implementation of the Library Elements Specific to Different Simulation Tools

The last step of the methodology for the design of co-simulation tools for continuous/discrete systems is the implementation of the library elements that are specific to different simulation tools. This step depends highly on the simulation tools chosen in the previous step, the analysis of the simulation tools.

3.2 Using Formal Methods for Co-Simulation Tools Design

This section gives more details on the steps that compose the generic stage (as presented in the previous section) as well as their implementation.

Before giving the details of a possible application of the methodology we present the basic concepts that are used in our specific methodology are introduced: Discrete Event System Specification (DEVS) [28], [29], timed automata [46], [47] and UPPAAL [48]. The following sub-sections present an example of utilization of the proposed methodology.

3.2.1 Basic Concepts

Discrete event system specifications

Discrete Event Systems Specifications (DEVS) is a formalism supporting a full range of dynamic system representation, with hierarchical and modular model development. The abstraction separates modeling from simulation and provides atomic models that can be used to build complex models that allow the integration of continuous and discrete-event models [28], [29]. It also provides all the mechanisms for the definition of an operational semantics for the continuous/discrete synchronization model, the high level representation of the global formal model.

A DEVS is defined as a structure [28], [29] :

$DEVS = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where

$X = \{(p_d, v_d) | p_d \in InPorts, v_d \in X\ p_d\ \}$ set of *input* ports and their values in the discrete event domain,

S - set of *sequential states*

$Y = \{(\ p_d, v_d) | p_d, \in OutPorts, v_d \in Y\ p_d\ \}$ set of *output* ports and their values in the discrete event domain.

$\delta_{int} : S \rightarrow S$ the *internal transition* function

δ_{ext}: $QxX \rightarrow S$ the *external transition* function, where:

$Q=\{(s,e)|s \in S, 0 \leq e \leq ta(s)\}$ set of total state,

e is the *time elapsed* since the last transition

$\lambda: S \rightarrow Y$ output function

t_a: $S \rightarrow R^+_{0,\infty}$ set of positive reals with 0 and ∞.

The system's state at any time is s**.** There are two possible situations:

- *case 1* – where we assume that no external events occur. In this case the system stays in this state s for the time $t_a(s)$. When the elapsed time e equals $t_a(s)$ (that is the time allocated for the system to stay in state s), the system outputs the value $\lambda(s)$. The state s changes to the state s' as a result of the transition $\delta_{int}(s)$. We emphasize here that the output is possible only before the internal transitions. We propose the definition of this type of transition using the following rule of the form $\frac{\textbf{Premises}}{\textbf{Conclusions}}$:

  $\frac{e = t_a(s) \wedge s' = \delta_{int}(s)}{(s,e) \xrightarrow{!\lambda(s)} (s',0)}$ where '!' represents the send operator.

- *case 2* – where there is an external event x before the expiration time, $t_a(s)$ (the system is in state (s,e), with $e \leq t_a(s)$), the system's state changes to state s' as a result of the transition $\delta_{ext}(s,e,x)$. For the definition of this type of transition, we propose the following rule:

  $\frac{e \leq t_a(s) \wedge s' = \delta_{ext}(s,e,x)}{(s,e) \xrightarrow{?x} (s',0)}$ where '?' represents the receive operator.

  Thus, the internal transition function dictates the system's new state when no external events occurred since the last transition while the external transition function dictates the system's new state when an external event occurs – this state is determined by the input x, the current state s and how long the system has been in this state, e. In both cases the system is then in some new state s' with some new expiration time $t_a(s')$.

We also give here DEVS coupled models as defined by the same formalism. For the case where we have ports, the specification includes external interfaces with input and output ports and values, and coupling relations.

$N = (X,Y,D, \{M_d | d \in D\}, EIC,EOC,IC)$ where:

$X = \{(p, v) | p \in InPorts, v \in Xp\}$ set of *input* ports and values,

$Y = \{(p, v) | p \in OutPorts, v \in Yp\}$ set of *output* ports and values

D = set of components names

$M_d = (X_d, S, Y_d, \delta_{int}, \delta_{ext}, \lambda, ta)$ is a DEVS with X_d , Y_d the set of input/output ports and values

EIC (External Input Coupling) = the coupling between the input in the coupled model and the external environment

EOC (External Output Coupling) = the coupling between the output from the coupled model and the external environment

IC (Internal Coupling) = the coupling between the modules that compose the coupled module

In our work we used the parallel DEVS coupled formalism. Each module composing the interface performs a different task accordingly to the continuous/discrete synchronization models.

Timed automata and UPPAAL

In this section we briefly introduce timed automata. A timed automaton [46] is a formalism for modeling and verification of real time systems. It can be seen as classical finite state automata with clock variables and logical formulas on the clock (temporal constraints) [47]. The constraints on the clock variables are used to restrict the behavior of the automaton. The logical clocks in the system are initialized to zero when the system is started and then increase at the uniform rate counting time with respect to a fixed global time frame. Each clock can be separately reset to zero. The clocks keep track of the time elapsed since the last reset [46]. There are two types of clock

constraints: constraints associated with transitions and constraints associated with locations. A transition can be taken when the clocks' values satisfy the *guard* labeled on it. Figure 3.6 illustrates an example of a timed automaton. The constraints associated with locations are called *invariants* and they specify the amount of time that may be spent in a location. The invariant "true" for a location means there are no constraints for the time spent in the location.

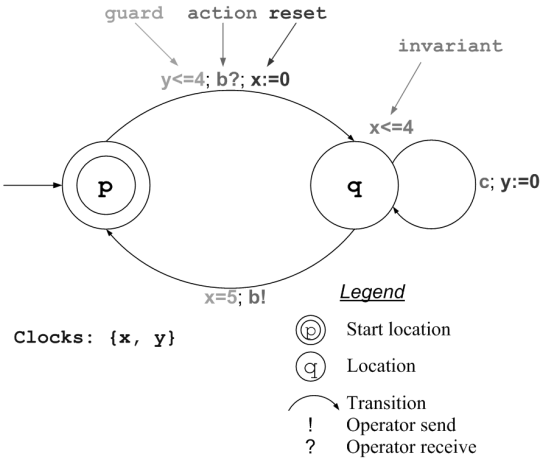


Figure 3.6. Example of a timed automaton

The process shown in Figure 3.6 starts at the location `p` with all its clocks (`x` and `y`) initialized to 0. The values of the clocks increase synchronously with time at the location `q`.

At any time, the process can change the location following a transition $p \xrightarrow{g;a;r} q$ if the current values of the clocks satisfy the enabling condition `g` (`guard`). A guard is a Boolean combination of integer bounds on clocks and clock-differences. With this transition, the variables are updated by `r` (`reset`) which is an action performed on clocks. The actions are used for synchronization and are expressed by `a` (`action`)[47]. A synchronization label is of the form *Expression?* or *Expression!* where `!` represents the operator `send` and `?` represents the operator `receive`.

The semantics for a time automaton are defined as "a transition system where a state or configuration consists of the current location and the current values of clocks" [47]. Thus, the state is represented by the tuple: (l, v) where l is the `location` and v is the `clock` valuation (a function that associates a real positive value, including zero, to each clock). Given the system, we can have two types of transitions between locations: a delay transition when the automaton may delay for some time or an action transition when the transition follows an enabled transition.

The transition showing the time passing is $(l,v) \xrightarrow{t} (l', v')$ if and only if:

$$\begin{cases} v' = v + t \\ \forall t' \in [0,t], (v + t') \text{ verifies } Inv(l) \end{cases}. \tag{8}$$

where *Inv(l)* is the *invariant* in the location *l, l=l', v'=v + t* showing that for all clocks *x, v'(x)=v(x)+t* .

For the discrete transitions $(p,v) \xrightarrow{g;a;r} (q,v')$ *v'* has to satisfy the *invariant* of *q*. *v'* is obtained from *v* by resetting the clocks indicated by the *reset r*.

Timed automata have the following characteristics that make them desirable for our formal model:

- Ease and flexibility of systems' modeling.
- Existence of a whole range of powerful tools that are already implemented and that allow different verification techniques.
- Adequate expressivity in order to model time constrained concurrent systems.

Our formal model needs to support concurrency between continuous/discrete systems thus it was represented as a parallel composition of several timed automata with no constraints regarding the time spent in the locations.

UPPAAL [48] is an integrated tool environment for modeling, simulation and verification of timed automata developed jointly by Aalborg University in Denmark and the Uppsala University in Sweden. It consists of three parts: a model descriptor, a simulator and a model-checker. The descriptor models systems that can be represented as a collection of non-deterministic processes with finite control structure and real-

valued clocks (i.e. timed automata), communicating through channels and (or) shared data structures. A model consists of one or more concurrent processes (also named here simulators), local and global variables, and channels. There are three types of locations in UPPAAL: *normal locations* with or without invariants, *urgent locations* and *committed locations*. No delay is allowed in urgent or committed locations. The transitions out from an urgent location have higher priority than that of time progress.

The expressions cover clocks and integer variables and are used with the labels: guards, synchronization, assignments or invariant. The models synchronize with each other via channels. In UPPAAL the assignments are evaluated sequentially (not concurrently). On synchronizing transitions, the assignments on the !-side (the emitting side) are evaluated before the ?-side (the receiving side).

The model checker engine in UPPAAL is based on the theory of timed automata and the query language is a subset of computational tree logic, the timed computational tree logic (TCTL). The query language [48] consists in path formulae and state formulae. The states formulae describe individual states while the path quantifies over traces of the model.

The main advantage of UPPAAL is that the product automaton is computed on-the-fly during verification. This reduces the computation time and the required memory space. It also allows interleaving of actions as well as hand-shake synchronization. In our approach UPPAAL was used for the formal representation of the simulation interfaces.

3.2.2 Definition of the Operational Semantics for the Synchronization in Continuous/Discrete Global Execution Models

DEVS allows for the definition of the operational semantics of the behavior of the co-simulation interfaces with respect to the synchronization models presented in Chapter 2.

Definition of the Operational Semantics for the canonical synchronization in Continuous/Discrete Global Execution Models

The operational semantics for the continuous/discrete canonical synchronization model is given by the set of rules presented in Table 3.1. *DataToCSI* (also *DataFromDSI*) is the output function from the discrete domain simulation interface $\lambda(s_d)$, and *DataToDSI* (also *DataFromCSI*) is the output function from the continuous domain interface $\lambda(s_c)$. The semantics of the global variable *flag* is related to the context switch between the continuous and discrete simulators. When *flag* is set to '1', the discrete simulator is executed. When it is '0', the continuous simulator is executed. The global variable *synch* is used to impose the order of the different operations expressed by the rules.

For a better explanation, we present in detail the first rule, corresponding to arrow 1 in Figure 2.3(a) and Figure 2.3(b). The premises of this rule are: the variable *synch* has the value '1', the variable *flag* has the value '1', and we have an external transition function (δ_{ext}) for the continuous model. The discrete model is initially in the total state (s_{dk}, e_{dk}), this means it has been in the state s_{dk} for the time e_{dk} . In this state, the discrete simulator performs the following actions:

- send the data and the value of its next time stamp (this action is expressed by *!(DataToCSI,* $t_a(s_{dk})$*)*
- switch the simulation context to the continuous model (this action is expressed by *flag = 0*).

For the same rule, the continuous model is in state q_k and performs the following actions:

- receive the data and the value of the time stamp from the discrete simulator (expressed by *?((DataFromDSI,* $t_a(s_{dk})$*).*
- set the global variable *synch* to '0' (action expressed by *synch=0*) in order to respect the premise of the rule corresponding to the arrow 4.

The actions expressed by this rule will be executed by the discrete simulator when the context will be switched to it.

Table 3.1. Operational semantics for the C/D canonical synchronization model

Rule	**- Arrows in Figure 2.3** **- Description**
$\dfrac{synch = 1 \wedge flag = 1 \wedge q_k = \delta_{ext}(q_k)}{(s_{dk}, e_{dk}) \xrightarrow{!(DataToCSI, t_a(s_{dk})); flag:=0} (s_{dk}, e_{dk}); q_k \xrightarrow{?(DataFromDSI, t_a(s_{dk})); synch:=0} q_k}$	- Arrow 1 fig. 2.3(a) and 2(b) - Context switch discrete to continuous
$\dfrac{flag = 0 \wedge \neg stateevent(t) \wedge q_{k+1} = \delta_{int}(q_k)}{q_k \xrightarrow{\delta_{int}} q_{k+1} \xrightarrow{!DataToDSI; flag:=1} q_{k+1}}$	- Arrow 2 and 3 in fig 2.3(a) - Continuous time advance and context switch continuous to discrete when no state event
$\dfrac{synch = 0 \wedge flag = 1 \wedge \neg stateevent \wedge s_{d(k+1)} = \delta_{ext}(s_{dk})}{(s_{dk}, e_{dk}) \xrightarrow{t_a(s_{dk}) - e_{dk}} (s_{dk}, t_a(s_{dk})) \xrightarrow{?DataFromCDI; \delta_{int}(s_{d(k+1)}); \lambda(s_{d(k+1)}); synch:=1} (s_{d(k+1)}, 0)}$	- Arrow 4 in figure 2.3(a) - Discrete time advance when no state event
$\dfrac{flag = 1 \wedge stateevent \wedge q_{k+1} = \delta_{int}(q_k)}{q_k \xrightarrow{!DataToDSI} q_{k+1} \xrightarrow{!DataToDSI; !t_{se}; flag:=1} q_{k+1}}$	- Arrow 2 and 3 in fig 2.3(b) - Continuous time advance and context switch continuous to discrete when state event
$\dfrac{synch = 0 \wedge flag = 1 \wedge stateevent \wedge s_{d(k+1)} = \delta_{ext}(s_{dk}, t)}{(s_{dk}, e_{dk}) \xrightarrow{?t_{se}} (s_{dk}, t_{se}) \xrightarrow{?DataFromCSI; \delta_{int}(s_{se}); \lambda(s_{se}); synch:=1} (s_{se}, 0)}$	- Arrow 4 fig. 2.3(b) - Discrete time advance when state event

Definition of the operational semantics for rollback-based synchronization in continuous/discrete global execution models

The operational semantics for the light rollback synchronization model is given by the set of rules presented in Table 3.2. This table respects the notations used for the canonical synchronization model presented in the section above. The semantic of the global variable *flag* is again related to the context switch between the continuous and discrete simulators. When *flag* is set to '1', the discrete simulator is executed. When it is '0', the continuous simulator is executed. For the rollback-based synchronization model, besides the global variable *synch* we introduce a new global variable *back*. These variable are used to impose the order of the different operations expressed by the rules (i.e when *back* is 1 the discrete simulator advances to the next time stamp while when it is 0, it backtracks to the previous time stamp).

For a better explanation, we detail here the first rule, corresponding to the arrow 1 in Figure 2.4. The premises of this rule are: the variables *synch, flag* and *back* have the value '1', and we have an external transition function (δ_{ext}) for the continuous model.

The discrete model is initially in the total state (s_{dk}, e_{dk}), this means it is in the state s_{dk} for the time e_{dk}. s_d is the tuple (x_{dk}, t_k). In this state, the discrete simulator performs the following actions:

- send the data and the value of its next time stamp (this action is expressed by *!(DataToCSI,* $t_a(s_{dk})$*)*
- switch the simulation context to the continuous model (this action is expressed by *flag = 0*).

Table 3.2. Operational semantics for the C/D rollback-based synchronization model

Rule	- Arrows in Figure 2.4 - Description
$\dfrac{synch=1 \wedge flag=1 \wedge back=1 \wedge s_{d(k+1)}=\delta_{int}(s_{dk})}{(s_{dk},e_{dk}) \xrightarrow{\delta_{int}(s_{dk})} (s_{d(k+1)},0) \xrightarrow{!(DataToCSI,t_a(s_{dk})); flag:=0} (s_{d(k+1)},t_a(s_{dk}))}$	- Arrow 1 in Figure 2.4(a) and 2.4(b) - Discrete time advance
$\dfrac{synch=1 \wedge flag=0 \wedge q_k=\delta_{ext}(q_k)}{q_k \xrightarrow{?(DataFromCSI,t_a(s_{dk})); synch:=0} q_k}$	- Arrow 2 in Figure 2.4(a) and (b) - Context switch discrete to cont.
$\dfrac{synch=0 \wedge flag=0 \wedge back=1 \wedge \neg statevent(t) \wedge q_{k+1}=\delta_{int}(q_k)}{q_k \xrightarrow{\delta_{int}} q_{k+1} \xrightarrow{!DataToDSI; flag:=1} q_{k+1}}$	- Arrows 3 and 4 Figure 2.4(a) -Cont. time advance and context switch cont. to discrete when no state event
$\dfrac{synch=0 \wedge flag=1 \wedge back=1 \wedge \neg statevent \wedge s_{d(k+1)}=\delta_{ext}(s_{d(k+1)})}{(s_{d(k+1)},t_a(s_{dk})) \xrightarrow{?DataFromCSI;\lambda(s_{d(k+1)});synch:=1} (s_{d(k+1)},0)}$	- Arrow 4 receiving end DSI fig. 2.4(a) - Context switch cont. to discrete when no state event
$\dfrac{synch=0 \wedge flag=0 \wedge back=1 \wedge statevent \wedge q_{(k+1)}=\delta_{int}(q_k)}{q_k \xrightarrow{\delta_{int}(q_k)} q_{(k+1)} \xrightarrow{!DataToDSI;!t_{se};flag:=1} q_{(k+1)}}$	- Arrow 3 and 4 Figure 2.4(b) - Cont. time advance and context switch cont. to discrete when state event
$\dfrac{synch=0 \wedge flag=1 \wedge back=1 \wedge statevent \wedge s_{d(k+1)}=\delta_{ext}(s_{d(k+1)},t)}{(s_{d(k+1)},e_{d(k+1)}) \xrightarrow{?DataFromCSI;?t_{se};\lambda(s_{d(k+1)});synch:=1;back:=0} (s_{d(k+1)},0)}$	- Arrow 4 receiving end DSI fig. 2.4(b) - Context switch cont. to discrete when state event
$\dfrac{\boldsymbol{synch=1 \wedge flag=1 \wedge back=0 \wedge s_{dk}=\delta_{int}(s_{d(k+1)})}}{\boldsymbol{(s_{d(k+1)},e_{d(k+1)}) \xrightarrow{\delta_{int}(s_{d(k+1)});back:=1} (s_{dk},e_{dk})}}$	- Arrow 5 Fig. 2.4(b) - Rollback in the discrete domain
$\dfrac{synch=1 \wedge flag=1 \wedge back=1 \wedge s_{se}=\delta_{int}(s_{dk})}{(s_{dk},t_{se}) \xrightarrow{\delta_{int}(s_{dk})} (s_{se},0) \xrightarrow{!(DataFromBus,t_a(s_{se})); flag:=0} (s_{d(k+1)},t_a(s_{se}))}$	- Arrow 6 and 7 Figure 2.4 (b) - Discrete time advance when state event

3.2.3 Distribution of the Synchronization Functionality to the Co-Simulation Interfaces

The second step of the methodology consists in the distribution of the synchronization functionality to the simulation interfaces. The synchronization functionality was presented in Section 3.1.3. Only the discrete domain interface changes with the synchronization model. This sub-section will present the two discrete simulation interfaces and the continuous domain interface. Before giving the distribution of the synchronization functionality in the co-simulation interfaces we present, for a better understanding of the notations used further, the global formal execution model.

The continuous/discrete global formal model

The global model proposed is formed by four sub-models (processes): the continuous domain simulator (Cont), the continuous simulation interface (CSI), the discrete domain simulator (Disc) and the discrete simulation interface (DSI).

Figure 3.7 shows the global formal model including the continuous domain and the discrete domain simulators and their interaction. The transitions show the synchronizations between the simulators and interfaces as well as the synchronization between the interfaces.

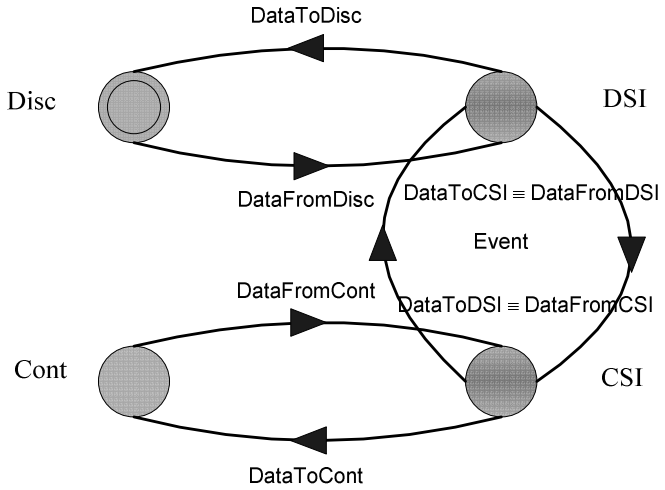


Figure 3.7. The global formal simulation model

The initial location for the global formal model is the discrete simulator; however, the continuous simulator is the first that advances in time. For a better understanding of the behavior of the simulation interfaces we used the following name conventions:

DataFromDisc - Data sent by the discrete simulator (Disc) to the discrete simulation interface (DSI)

DataToDisc – Data sent by DSI to Disc

DataFromCont - Data sent by the continuous simulator (Cont) to the continuous simulation interface (CSI)

DataToCont - Data sent by CSI to Cont

DataToCSI – Data sent by the discrete simulation interface (DSI) to the continuous simulation interface (CSI)

DataFromDSI – Data received by CSI from DSI

One can observe that *DataToCSI* and *DataFromDSI* are the same but for an ease in understanding the rules that will be presented in the following sections we will use both notations: *DataToCSI* for the representation from the discrete simulation interface point of view and *DataFromDSI* for the representation from the continuous simulator point of view.

DataToDSI – Data sent by CSI to DSI

DataFromCSI – Data received by DSI from CSI

In this case we make the same comment – *DataToDSI* and *DataFromCSI* are the same but for the reason presented above we will use both notations.

The discrete domain simulation interface for the canonical synchronization model

This section presents the behavior and the operational semantics of the Discrete Simulation Interfaces (DSI). The behavior of the discrete domain interface can be described by a few processing steps detailed in Figure 3.8.

The interface is in charge of:

- exchanging *data* between the simulators (send/receive),

- sending the *time stamps* of the next events,
- considering the *state events* and
- the *context switch* to the continuous interface.

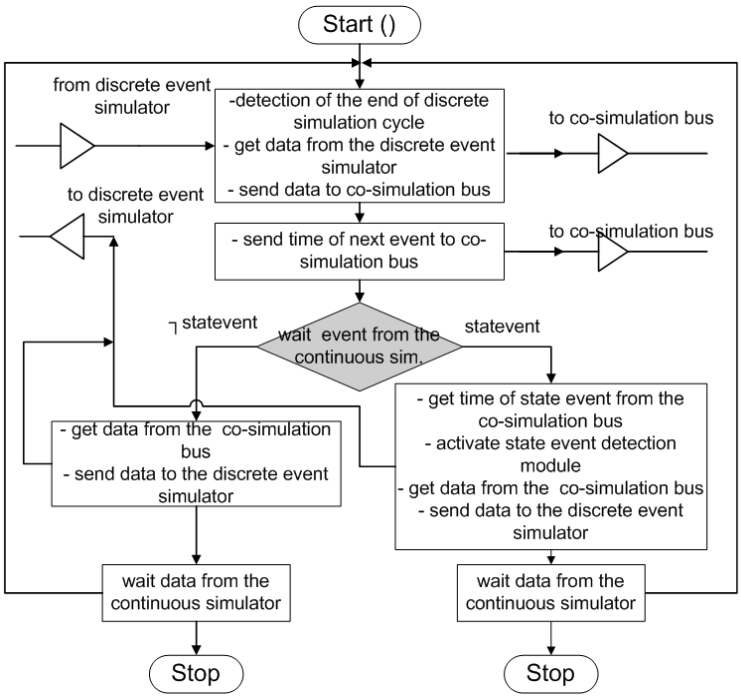


Figure 3.8. Flowchart for the discrete domain interface for the canonical synchronization model

More detailed, after starting, the tasks of the interface are:

- get data from the discrete simulator. This data is sent to the co-simulation bus
- detect the end of discrete simulation cycle. The time of the next event is sent to the co-simulation bus.
- wait for event from the co-simulation bus. If a state event was generated, the interface gets the time of the state event and the data from the co-simulation bus and sends them to the discrete simulator. If no state event was generated the interface sends to the discrete simulator, only the data from the continuous interface. Only now the time in the discrete simulator advances to the next event/state event

- wait for data the continuous simulator - the cycle restarts.

The semantics was defined using DEVS formalism. Table 3.3 presents a set of rules that show the transition between states. The first rule covers arrow 1 in Figure 2.3(a) and Figure 2.3(b). The second and third rules correspond to arrows 3 (on the receiving part) and 4 in Figure 2.3(a) respectively Figure 2.3(b).

Table 3.3. Operational semantics for the Discrete Simulation Interface (DSI) for the canonical synchronization model

Rule
$(1)\ \dfrac{synch = 1 \wedge flag = 1 \wedge s_{dk} = \delta_{ext}(s_{dk}, 0, x)}{(s_{dk}, \infty) \xrightarrow{?\,DataFromDisc} (s_{dk}, 0) \xrightarrow{!(data, t_{k+1}((s_{dk}))); flag := 0} ((s_{dk}), t_{d(k+1)})}$
$(2)\ \dfrac{synch = 0 \wedge flag = 1 \wedge \neg stateevent \wedge s_{dk} = \delta_{ext}(s_{dk}, 0, x)}{(s_{dk}, e_{dk}) \xrightarrow{?\,Event} (s_{dk}, 0) \xrightarrow{?data; synch := 1} (s_{dk}, 0) \xrightarrow{!\,DataToDisc} (s_{d(k+1)}, e_{dk+1})}$
$(3)\ \dfrac{synch = 0 \wedge flag = 1 \wedge stateevent \wedge (s_{dk}) = \delta_{ext}((s_{dk}), 0, x)}{(s_{dk}, e_{dk}) \xrightarrow{?\,Event} (s_{dk}, 0) \xrightarrow{?(data, tse); synch := 1} (s_{dk}, 0) \xrightarrow{!(DataToDisc, tse)} (s_{se}, e_{se})}$

In order to clarify, we detail here the first rule. The premises of this rule are: the *synch* variable has value '1', the *flag* variable has value '1', and we have an external transition function (δ_{ext}) for the DSI.

This rule expresses the following actions of the discrete simulator interface:

- receiving data from the discrete model. This is an external transition (δ_{ext}) expressed by *?(DataFromDisc)*.
- sending data to the Continuous Simulator Interface (CSI) (*!DataToCSI*). The data sent to the CSI is the output function $\lambda(s_{dk})$ and it is possible, according with DEVS formalism, only as a consequence of an internal transition (δ_{int}). In our case the output is represented by $!(data, t_{d(k+1)}(s_{dk}))$. This transition corresponds to arrow 1 in Figure 2.3(a) and 2.3(b).

- switching the simulation context from the discrete to the continuous domain (action expressed by *flag:=0*).

All the other rules presented in this table follow the same format.

From these rules we can trace the state graph of the DSI for the canonical synchronization model as shown in Figure 3.9. The dashed lines represent internal transitions and the corresponding states and the plain lines represent external transitions and the corresponding states.

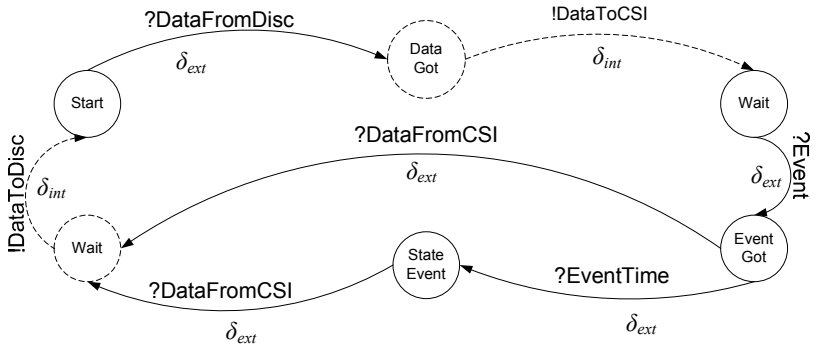


Figure 3.9. State graph of the DSI for the canonical synchronization model represented using DEVS

The discrete domain simulation interface for the rollback-based synchronization model

The behavior of the discrete domain interface in the case of the rollback-based synchronization model can be described by a few processing steps detailed in Figure 3.10.

The interface is in charge of:

- exchanging *data* between the simulators (send/receive),
- sending the *time stamps* of the next events,
- advancing the *time* to the next discrete event
- restoring the previous *state* if a state event was generated by the continuous simulator
- considering the *state events* and

- the *context switch* to the continuous interface.

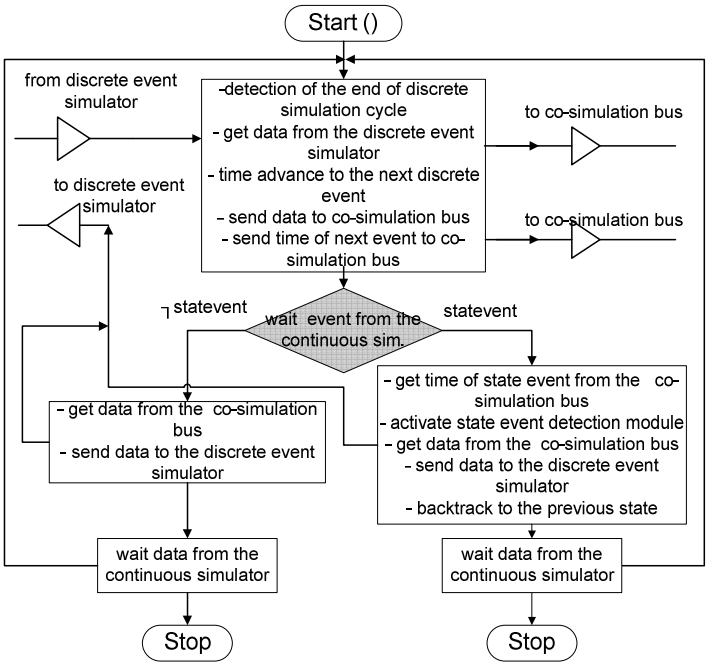


Figure 3.10. Flowchart for the discrete domain interface for the rollback-based synchronization model

More detailed, after starting, these tasks are:

- get data from the discrete simulator. This data is sent to the co-simulation bus
- detect the end of discrete simulation cycle. The time of the next event is sent to the co-simulation bus. Time advances to the next discrete event.
- wait for event from the co-simulation bus. If a state event was generated, the interface gets the time of the state event and the data from the co-simulation bus and sends them to the discrete simulator. The discrete simulator backtracks to the previous stable state. If no state event was generated the interface sends to the discrete simulator, only the data from the continuous interface.
- wait for data the continuous simulator - the cycle restarts.

Table 3.4 presents a set of rules that show the transition between states. Rule number 4 covers arrow 1 in both Figure 2.4(a) and 2.4(b). Rule 5 corresponds to the arrow 4 – no

state event- (on the receiving part) in Figure 2.4(a). Rules 6, 7 and 8 show the behavior for the discrete simulator backtracking and advancing to the state event time t_{se} (arrows 5, 6, 7 in Figure 2.4(b)).

Table 3.4. Operational semantics for the DSI for the rollback-based synchronization model

Rule
$(4)\ \dfrac{synch = 1 \wedge flag = 1 \wedge back = 1 \wedge s_{d(k+1)} = \delta_{int}(s_{dk})}{(s_{dk}, e_{dk}) \xrightarrow{?DataFromDisc} (s_{d(k+1)}, 0) \xrightarrow{!(DataToCSI, t_a(s_{dk}, t_k)); flag:=0} ((s_{d(k+1)}), t_a(s_{dk}))}$
$(5)\ \dfrac{synch = 0 \wedge flag = 1 \wedge back = 1 \wedge \neg statevent \wedge s_{d(k+1)} = \delta_{ext}(s_{d(k+1)})}{(s_{d(k+1)}, t_a(s_{dk})) \xrightarrow{?DataFromCSI; \lambda(s_{d(k+1)}); synch:=1} (s_{d(k+1)}, 0)}$
$(6)\ \dfrac{synch = 0 \wedge flag = 1 \wedge back = 1 \wedge statevent \wedge s_{d(k+1)} = \delta_{ext}(s_{d(k+1)}, t)}{(s_{d(k+1)}, e_{d(k+1)}) \xrightarrow{?DataToCSI; ?t_{se}; \lambda(s_{d(k+1)}); synch:=1; back:=0} (s_{d(k+1)}, 0)}$
$(7)\ \dfrac{synch = 1 \wedge flag = 1 \wedge back = 0 \wedge statevent \wedge s_{dk} = \delta_{int}(s_{d(k+1)})}{(s_{d(k+1)}, e_{d(k+1)}) \xrightarrow{\delta_{int}(s_{d(k+1)}); back:=1} (s_{dk}, e_{dk})}$
$(8)\ \dfrac{synch = 1 \wedge flag = 1 \wedge back = 1 \wedge statevent \wedge s_{se} = \delta_{int}(s_{dk})}{(s_{dk}, t_{se}) \xrightarrow{\delta_{int}(s_{dk})} (s_{dse}, 0) \xrightarrow{!(DataToCSI, t_a(s_{dse})); flag:=0} (s'_{d(k+1)}, t_a(s_{se}))}$

In order to clarify, we detail here the first rule. The premises of this rule are: the *synch* variable has value '1', the *flag* variable has value '1', the *back* variable has also value '1'and we have an external transition function (δ_{ext}) for the DSI. These variables insure the correct transitions for the simulator during the context switch. For example if there is no state event the next transition in the discrete domain is from time t_{k+1} to time t_{k+2} (the variables *synch, flag* and *back* have all value '1'). If the continuous simulator generates a state event the next transition in the discrete domain is from time t_{k+1} to time t_k (the arrow 5 in the Figure 2.4) and variables *synch* and *flag* have the value '1' while the variable *back* has the value '0'. This rule expresses the following actions of the discrete simulator interface:

- receiving data from the discrete model. This is an external transition (δ_{ext}) expressed by *?(DataFromDisc)*.
- sending data to the Continuous Simulator Interface (CSI) (*!DataToCSI)*. The data sent to the CSI is the output function $\lambda(s_{dk})$ and it is possible, according with DEVS formalism, only as a consequence of an internal transition (δ_{int}).
- switching the simulation context from the discrete to the continuous domain (action expressed by *flag:=0*).

All the other rules presented in this table follow the same format.

From these rules we can trace the state graph of the DSI for the canonical synchronization model as shown in Figure 3.11. The dashed lines represent internal transitions and the corresponding states and the plain lines represent external transitions and the corresponding states.

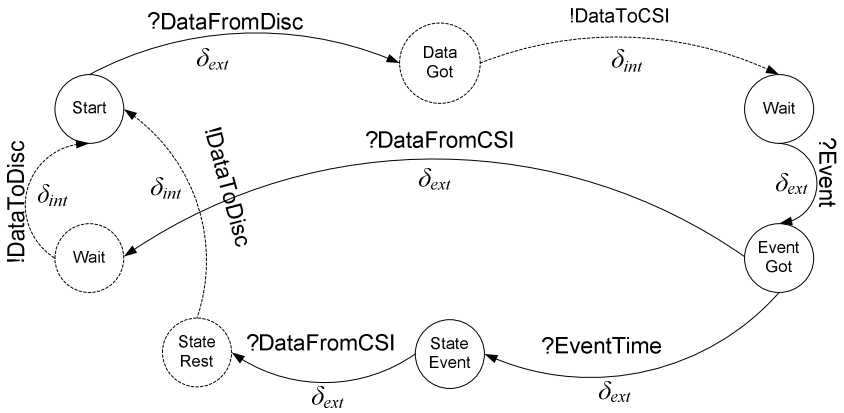


Figure 3.11. State graph of the DSI for the rollback-based synchronization model represented using DEVS

The continuous domain simulation interface

The behavior of the continuous domain interface can also be described by a few processing steps detailed in Figure 3.12.

This interface handles:

- exchanging *data* between the simulators (send/receive),
- sending the *time stamps* of the next events,

- the indication (to the discrete interface) of the occurrence of a *state event*, and
- the *context switch* to the discrete interface.

More detailed, after starting, the tasks of the continuous domain simulation interface are:

- *get data and time* of next discrete event from the co-simulation bus. This data is sent to the continuous simulator
- *get data* from the continuous simulator. If a state event was generated by the continuous simulator, the interface sends the time of the state event and the data to the co-simulation bus. If no state event is generated, the continuous interface sends only data to the co-simulation bus.
- *wait* for data/time from the discrete simulator; the cycle restarts.

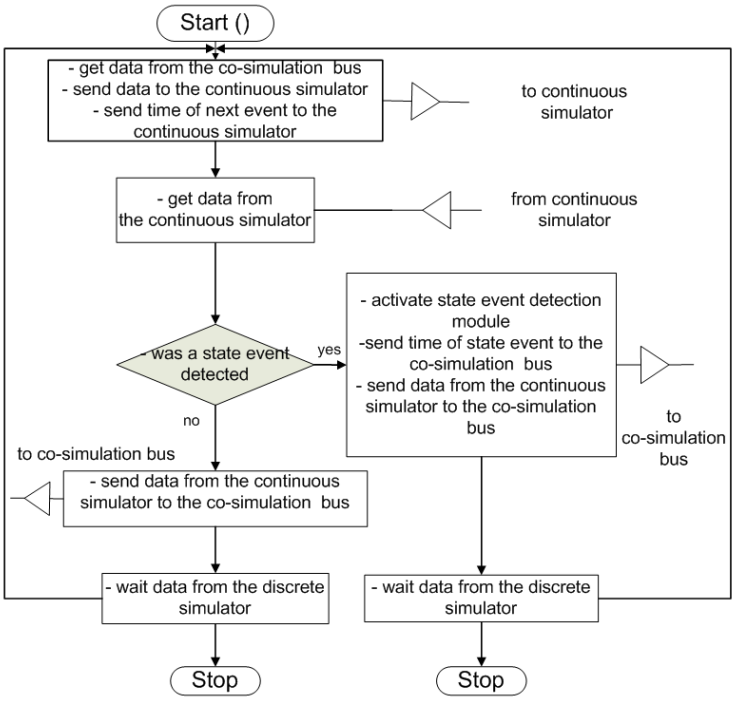


Figure 3.12. Flowchart for the continuous domain interface

The operational semantics for the CSI is given by the set of rules presented in Table 3.5. In these rules, the *data* notation refers to the data exchanged between the DSI and the discrete simulator.

Table 3.5. Operational semantics for the Continuous Simulation Interface (CSI)

Rule
$(9)\ \dfrac{synch = 1 \wedge flag = 1 \wedge q_k = \delta_{ext}(q_k),0,x)}{q_k \xrightarrow{?(data,t_{d(k+1)});synch:=0} q_k \xrightarrow{!(DataToCont,t_a(s_{dk}))} q_k}$
$(10)\ \dfrac{synch = 0 \wedge flag = 0 \wedge \neg stateevent \wedge q_{k+1} = \delta_{int}(q_k)}{q_k \xrightarrow{?(DataFromCont)} q_k \xrightarrow{!(data);flag:=1} q_{(k+1)}}$
$(11)\ \dfrac{synch = 0 \wedge flag = 0 \wedge stateevent \wedge q_{k+1} = \delta_{int}(q_k)}{q_k \xrightarrow{?(DataFromCont)} q_k \xrightarrow{!(data,t_{se});flag:=1} q_{se}}$

From these rules we can trace the state graph of the CSI as shown in Figure 3.13

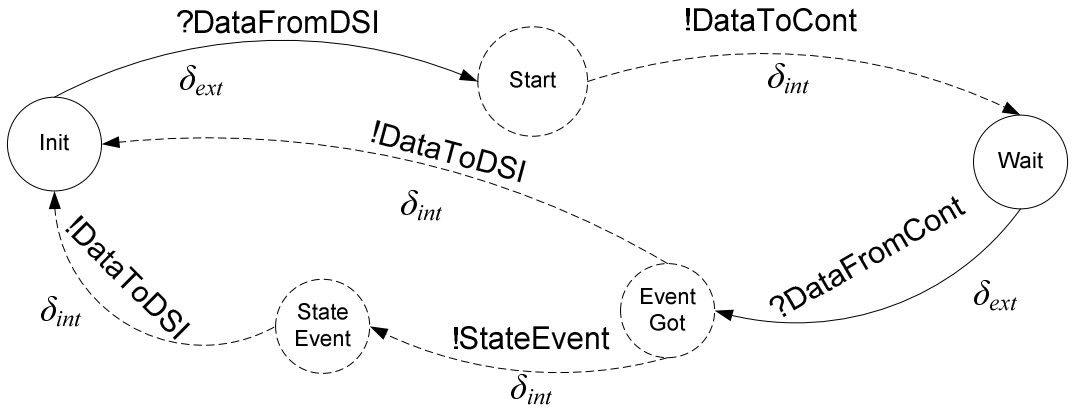


Figure 3.13. State graph of the CSI represented using DEVS

3.2.4 Formalization and Verification of the Co-Simulation Interfaces Behavior

In [49] the authors demonstrate the equivalence between a DEVS model and the timed automata. The timed-automata model completes the DEVS graph with the addition of

the timing evolution notions. In this work, both models of synchronization were formalized and verified. The continuous domain simulation interface is the same for both models. This sections present the formal representation for each discrete interface as well as for the continuous interface.

The formalization of the discrete domain simulation interface for the canonical synchronization

Figure 3.14 shows the formal model for the discrete domain interface, using timed automata. The model has only one initial location (marked in Figure 3.14 by a double circle) *Start*.

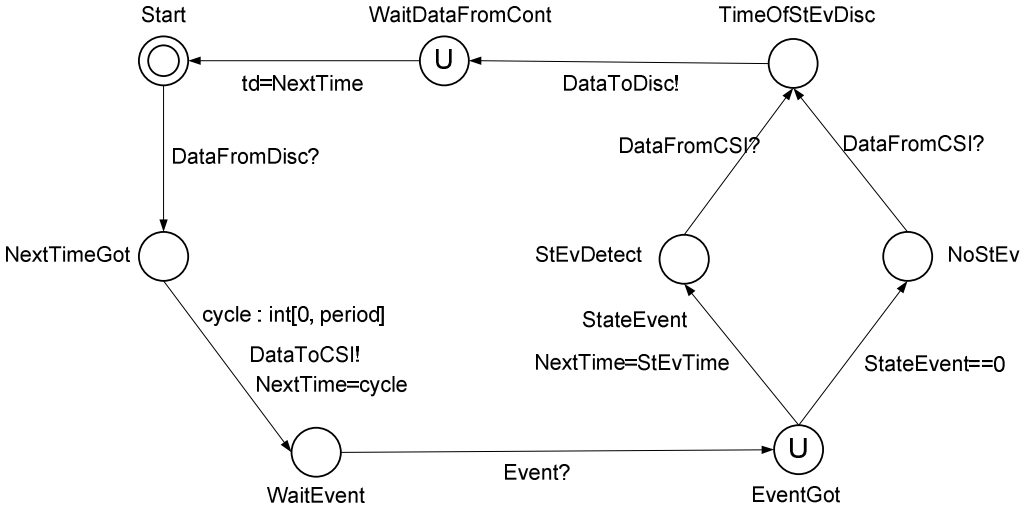


Figure 3.14. The DSI for the canonical synchronization model represented as a timed automaton

The discrete interface will change location from *Start* to *NextTimeGot* following the transition $Start \xrightarrow{DataFromDisc?} NextTimeGot$. This is an external transition realized in zero time and it is triggered by receiving the data (that is also synchronization between the discrete simulator and the interface) from the discrete simulator (`DataFromDisc?`). Here the interface receives the data from the discrete simulator and the time of the next event in the discrete domain.

The location changes to *WaitEvent* following the transition:

$NextTimeGot \xrightarrow{DataToCSI!,\ NextTime=cycle,\ cycle:int[0,period]} WaitEvent$

In order to change the location, the continuous interface sends the time of the next event (occurred/scheduled event) in discrete (the synchronization `DataToCSI!`) to the discrete interface. The variable `NextTime` is the time of the next event in the discrete domain. This variable takes, in this mode, the value `cycle`. The theory normally assumes equidistant sampling intervals. This assumption is not usually achieved in practice. For an accurate simulation we assume that cycle takes random values in an interval defined here as *[0, period]*. In *WaitEvent* location, the context is switched from the discrete to the continuous simulator.

When the context is switched back to the discrete simulator, the location is changed to *EventGot* following the synchronization transition: $WaitEvent \xrightarrow{Event?} EventGot$. During this transition the discrete interface receives from the continuous interface the synchronization `Event?`. In this location the occurrence of a state event in the continuous domain is considered. *EventGot* is an urgent location (as defined in section 3.2.1). This will not allow the discrete model to miss a state event generated by the continuous model. Two cases are possible:

- When no state event was generated by the continuous domain, the location changes from *EventGot* to *NoStEv*. The transition $EventGot \xrightarrow{StateEvent==0} NoStEv$ is annotated in this case only with the guard `StateEvent==0`.

- When a state event was generated by the continuous domain the location changes from *EventGot* to *StEvDetect* following the transition:

$\boldsymbol{EventGot \xrightarrow{StateEvent,\ NextTime=StEvTime} StEvDetect}$.

This transition is annotated with a guard (`StateEvent`) and the update of the `NextTime` in the discrete domain as the time when the state event occurred in the continuous domain `StEvTime` (for a rigorous synchronization, the discrete domain has to consume this event and stop at the time when it was generated by the continuous domain interface). This is the time of the next event that is going to be sent to the

continuous simulator. From both locations *StEvDetect* and *NoStEv*, the system can reach the next location: *TimeOfStEvDisc*. In both cases the model performs synchronization (`DataFromCSI?`). At this point the discrete interface will synchronize and send data to the discrete simulator (`DataToDisc!`) and changes the location to *WaitDataFromCont*. The next location is *Start*, the discrete time variables is initialized on this channel (`td=NextTime`) and the cycle restarts.

The formalization of the discrete domain simulation interface for the rollback-based synchronization

Figure 3.15 shows the formal model (using timed automata) for the discrete domain interface. The model has only one initial location (a double circle in Figure 3.15) *Start*. The discrete interface will change location from *Start* to *NextTimeGot* following the transition $Start \xrightarrow{DataFromDisc?} NextTimeGot$. This is an external transition realized with zero time and it is triggered by the receiving of the data (that is also synchronization between the discrete simulator and the interface) from the discrete simulator (`DataFromDisc?`). Here the interface receives the data from discrete simulator and the time of the current event in the discrete domain.

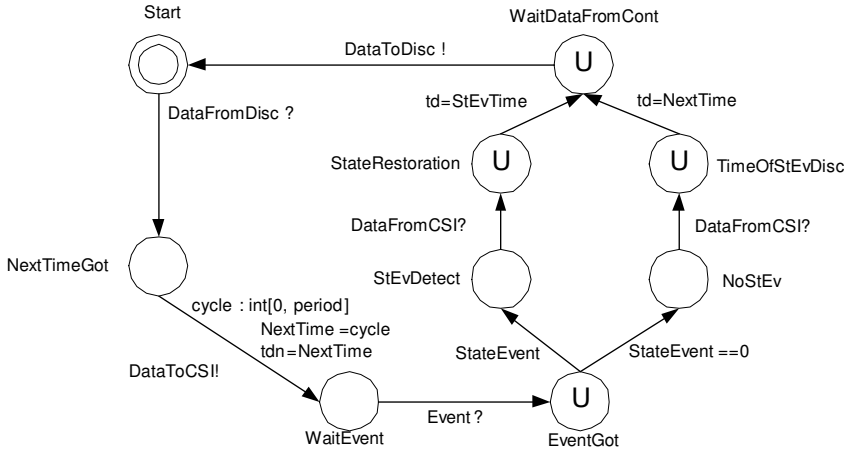


Figure 3.15. The DSI for the rollback-based synchronization model represented as a timed automaton

The location changes then to *WaitEvent*. The discrete interface sends to the continuous interface the time of the current event (the synchronization `DataToCSI!`). The variable `NextTime` represents the time of the events in the discrete domain. This variable takes the value `cycle`. This values is then assigned to the variable `tdn` that represents the time stamp of the event. The theory normally assumes equidistant sampling intervals. This assumption is not usually achieved in practice. For an accurate simulation we assume that cycle takes random values in an interval defined here as *[0, period]*. In *WaitEvent* location, the context is switched from the discrete to the continuous simulator. When the context is switched back to the discrete simulator, the location is changed to *EventGot* following the synchronization transition: $WaitEvent \xrightarrow{Event?} EventGot$. During this transition the discrete interface receives from the continuous interface the synchronization `Event?`. In this location the occurrence of a state event in the continuous domain is considered. *EventGot* is an urgent location. This will not allow the discrete model to miss a state event generated by the continuous model. Two cases are possible:

- When no state event was generated by the continuous domain, the location changes from *EventGot* to *NoStEv*. The transition $\boldsymbol{EventGot \xrightarrow{StateEvent==0} NoStEv}$ is annotated in this case only with the guard `StateEvent==0`. This state changes to *TimeOfStEvDisc* (that is an urgent location) following the transition $\boldsymbol{NoStEv \xrightarrow{DataToBus?} TimeOfStEvDisc}$. This is an external transition realized with zero time and it is triggered by the receiving of the data (that is also synchronization between the discrete and the continuous interfaces) from the continuous interface (`DataFromCSI?`). During this transition the data from continuous discrete simulator. The system will immediately change the state to *WaitDataFromCont* while updating the time in discrete with the time stamp of the current event (`td=NextTime`).
- When a state event was generated by the continuous domain the location changes from *EventGot* to *StEvDetect* following the transition: $\boldsymbol{EventGot \xrightarrow{StateEvent} StEvDetect}$. This

transition is annotated with a guard (`StateEvent`). This state changes to *StateRestoration* following the transition $StEvDetect \xrightarrow{DataFromCSI?} StateRestoration$. This is also an external transition realized with zero time. During this transition only the data is sent to the discrete simulator. The system will immediately change the state to *WaitDataFromCont* while updating the time in discrete with the time stamp of the state event (`td=StEvTime`).

From *WaitDataFromCont* state the location changes to *Start*. The discrete interface sends to the discrete simulator the data and the time of the events (state event or discrete event) and is represented here by the synchronization `DataToDisc!` and the cycle restarts.

The formalization of the continuous domain simulation interface

Figure 3.16 shows the formal model (using timed automata) for the continuous domain interface. The model also has only one initial location (marked in Figure 3.16 by a double circle) *Start*.

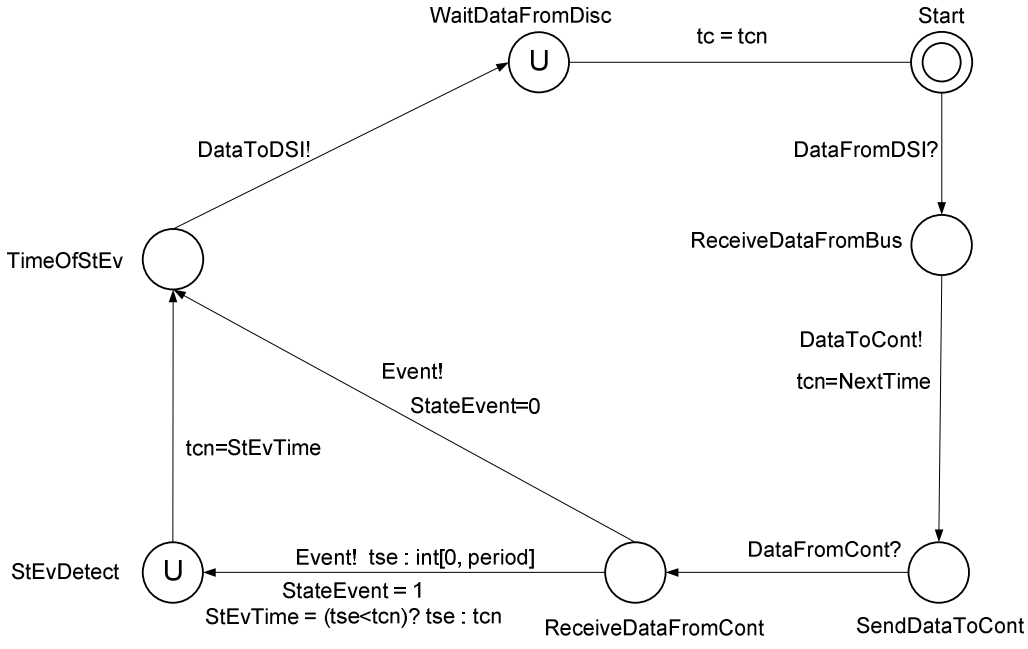


Figure 3.16. The CSI represented as a timed automaton

The continuous interface will leave the initial location *Start* following the transition: $Start \xrightarrow{DataFromDSI?} ReceiveDataFromBus$. This is also an external transition realized with zero time and it is triggered by the reception of the data from the discrete interface (`DataFromDSI?`) that is also the first synchronization point between the discrete interface and the continuous interface. The interface receives the data from the discrete simulator and the time of the next event in the discrete model. From the *ReceiveDataFromBus* location the process moves to the next location *SendDataToCont* following the transition

$$ReceiveDataFromBus \xrightarrow{DataToCont\ !,\ tcn=NextTime} SendDataToCont$$

The value `NextTime`, the time of the next event (occurred/scheduled event) in the discrete simulator is assigned to `tcn`, the next time in the continuous simulator. In our model, the synchronization on this transition is between CSI and Cont (where Cont is the continuous domain simulator), the interface sends data received from DSI and the time of the next event in the discrete domain to the simulator.

The system changes the location from *SendDataToCont* to *ReceiveDataFromCont* following the synchronization transition: $SendDataToCont \xrightarrow{DataFromCont?} ReceiveDataFromCont.$ During this transition the continuous interface receives data from the continuous simulator and, if a state event occurred, the time of the state event. In the *ReceiveDataFromCont* location, the continuous interface evaluates if a state event was generated. Two cases are possible:

- When no state event is generated, the location changes from ReceiveDataFromCont to TimeOfStEv following the transition $ReceiveDataFromCont \xrightarrow{Event!StateEvent=0} TimeOfStEv$. The transition is annotated in this case by the synchronization Event! and with the update StateEvent=0.
- When a state event is generated, the location changes from *ReceiveDataFromCont* to *StEvDetect* following the transition:

$$ReceiveDataFromCont \xrightarrow{Event!\ StateEvent=1,StEvTime=(tse<tcn)?tse:tcn,\ tse:int[0,period]} StEvDetect$$

This transition is annotated with a synchronization (`Event!`) and three variable updates: `StateEvent=1` (for the detection of a state event), `StEvTime=(tse<tcn)? tse:tcn, tse:int[0,period]` (for the time of the state event that occurs during the time interval *[0,period]*; this time will be sent to the discrete simulator). *StEvDetect* is an urgent location. The location *StEvDetect* changes to *TimeOfStEv* following the transition $StEvDetect \xrightarrow{tcn=StEvTime} TimeOfStEv$.

At this point there is no synchronization, only an update of the time in the continuous domain having assigned the time of the state event `StEvTime: tcn=StEvTime`.

TimeOfStEv location is common for both cases, `StateEvent=0` or `StateEvent=1`. This location changes to *WaitDataFromDisc*. The system performs synchronization (`DataToDSI!`) between the continuous interface and the continuous simulator. The next location is *Start*, the continuous time variables is initialized on this channel (`tc=tcn`) and the cycle restarts.

Formal model simulation

The UPPAAL tool allows the validation of the system's expected behavior regarding functionality: synchronization, conflicts, and communication. We simulated all the possible dynamic executions of our model. Figure 3.17 shows a screenshot of the simulator.

We observe that the left panel is the simulation control window. It highlights the enabled transition as well as the symbolic traces. The middle panel shows the variables. It displays the values of the data and clock variables in the current location or transition selected in the trace of the simulation control panel (the symbolic traces). The right panel allows the visualization of the message sequence chart (also known as simulator). The vertical lines in the simulator window in Figure 3.17 represent the transitions between the locations while the horizontal lines are the synchronization points. In this figure the communication between the interfaces as well as the communication between

the simulators and the domain specific interfaces are represented by the same horizontal lines.

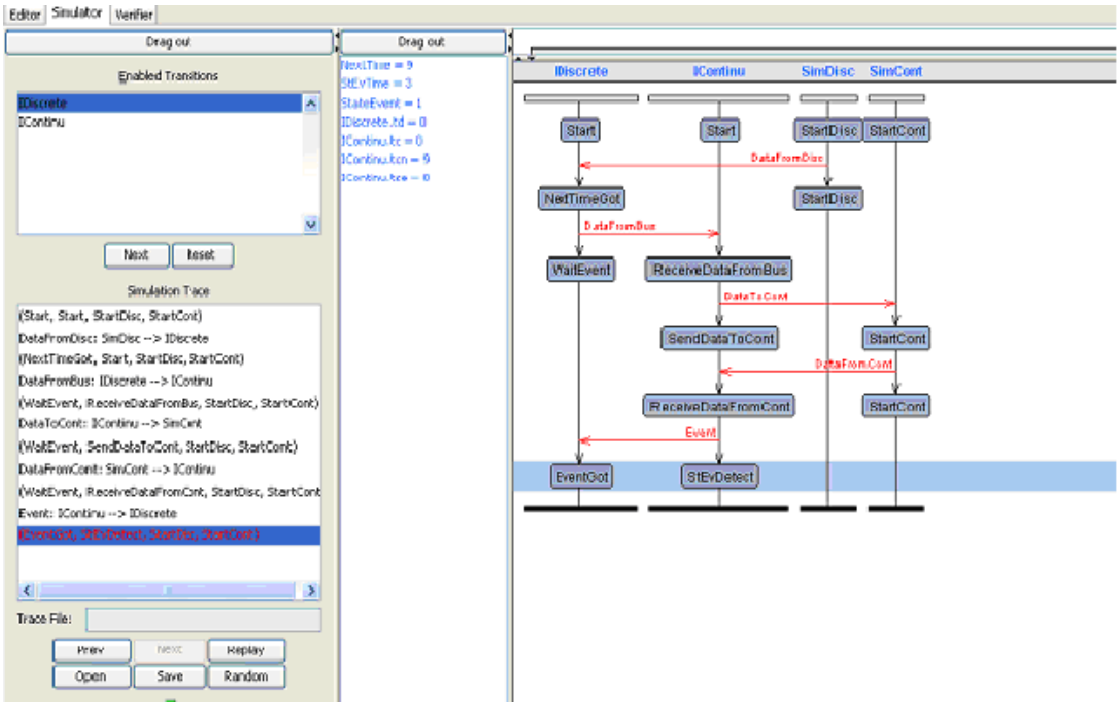


Figure 3.17. Formal model simulation screen capture

As shown here, the simulation was stopped by the user after the detection of a state event in the continuous domain. The state event was indicated to the discrete simulator and the time of the state event (StEvTime) is to be sent from the continuous to the discrete interface. The variable panel shows that the variable StateEvent=1, the time of the state event StEvTime=2, and the NextTime=10. The discrete simulator, instead of advancing the time to 10, will advance only to StEvTime.

The verification of the execution model

The formal verification consists of checking properties of the system for a broad class of inputs [44]. In our work we checked properties that fall into three classes:

- *Safety properties* - the system does not reach an undesirable configuration, (e.g., deadlock) [50].

- *Liveness properties* - some desired configuration will be visited eventually or infinitely (e.g., expected response to an input) [50].
- *Reachability properties* – the system always has the possibility of reaching a given situation (some particular situation can be reached) [44].

The properties verified in order to validate the synchronization models are described below. Properties *P0* to *P4* were checked for both synchronization models. Property *P5* was checked only for the canonical synchronization model because the backtracking in the rollback based synchronization model.

P0 *Absence of deadlock (safety property)*

Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set. In UPPAAL deadlock is expressed by a formula using the keyword `deadlock`. A state is a deadlock state if there are no outgoing action transitions either from the state itself or any of its delay successors [44].

```
A[] not deadlock
```

P1 *State event detected by the discrete domain (liveness property)*

The indication of a state event by the continuous interface and its detection by the discrete interface is very important for continuous/discrete heterogeneous systems. We defined a liveness property in order to check this behavior that is stated as follows:

Definition: A state event detected in the continuous domain `leads` to a state event detected in the discrete.

```
IContinu.StEvDetect --> IDiscrete.StEvDetect
```

P2 *No state event in discrete if no state event in continuous domain (safety property)*

In order to avoid false responses from the discrete simulators, we defined a safety property to verify if the system will "detect" a state event in the discrete simulator when it was not generated (and indicated) by the continuous domain:

Definition: `Invariantly` a state event detected in the discrete domain `imply` state event in the continuous.

```
A[](IDiscrete.StEvDetect imply StateEvent)
```

P3 Synchronization between the interfaces (reachability property)

One of the most important properties characterizing the interaction between the continuous and the discrete domains is the communication and implicitly the synchronization. This property verifies that after a cycle executed by each model, both are at the same time stamp (and by consequence are synchronized)

Definition: `Invariantly` both processes in the *Start* location (initial state) `imply` the time in the continuous domain t_c is equal with the time in the discrete domain t_d.

```
A[](  (IDiscrete.Start  and  IContinu.Start)  imply  (
IContinu.tc – IDiscrete.td <= period))
```

P4 Synchronization between the interfaces when a state event was detected (reachability property)

This property verifies that there is synchronization between the interfaces even when a state event is detected.

Definition: The discrete process in the *StateRestoration* location and the `continuous` process in the *StEvDetect* location leads to the time in the continuous t_c is equal with the time in the discrete t_d.

```
(IDiscrete.StateRestoration  and  IContinu.StEvDetect)  -->
(IContinu.tc- IDiscrete.td == 0)
```

P5 Causality principle (liveness property) – checked only for the canonical synchronization model

The causality can be defined as a cause and effect relationship. The causality of two events describes to what extent one event is caused by the other. The causality is already verified by ***P3*** for scheduled events. However, when a state event is generated by the continuous domain, the discrete domain has to detect this event at the same precise time (the cause precedes or equals the effect time) and not some other possible event existing at a different time in the continuous domain.

Definition: `Invariantly` both processes in the *StEvDetect* location (detection of state event) `imply` the time in the continuous t_c is equal with the time in the discrete t_d.

```
A[]( (IDiscrete.Start and IContinu.Start ) imply ( IContinu.tc - IDiscrete.td == 0))
```

3.2.5 Definition of the Internal Architecture of the Co-Simulation Interfaces

The overall continuous/discrete simulation interface is formally defined using the DEVS formalism. As shown in Figure 3.5, the interface is described as a set of coupled models: the continuous domain interface (CDI), the discrete domain interface (DDI) and the co-simulation bus. Figure 3.18 shows the atomic modules composing the interface used in our implementation.

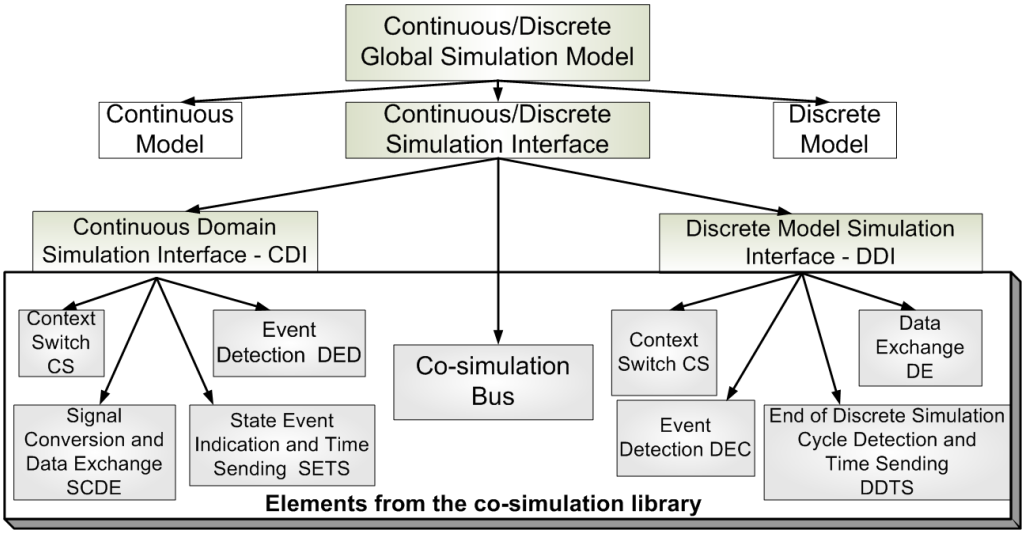


Figure 3.18. The hierarchical representation of the generic architecture of the co-simulation model with elements of the co-simulation library defined

The specific functionalities of the interfaces were presented in section 3.1.2. In terms of internal architecture, the blocks assuring these features are:

For the *Continuous Model Simulation Interface*

- The *State Event Indication and Time Sending* block (SETS)
- The *Signal Conversion and Data Exchange* block (SCDE)
- The *Event Detection* block (DED)

- The *Context Switch* block (CS)

For the *Discrete Model Simulation Interface*

- The *End of Discrete Simulation Cycle Detection and Time Sending* block (DDTS)
- The *Data Exchange* block (DE)
- The *Event Detection* block (DEC)
- The *Context Switch* block (CS)

These atomic modules are forming the co-simulation library and the co-simulation tools enable their parameterization and their assembly in order to generate a new co-simulation instance.

Figure 3.19 presents the atomic modules interconnection in each domain specific simulation interface as well as the signals and interactions between the interfaces.

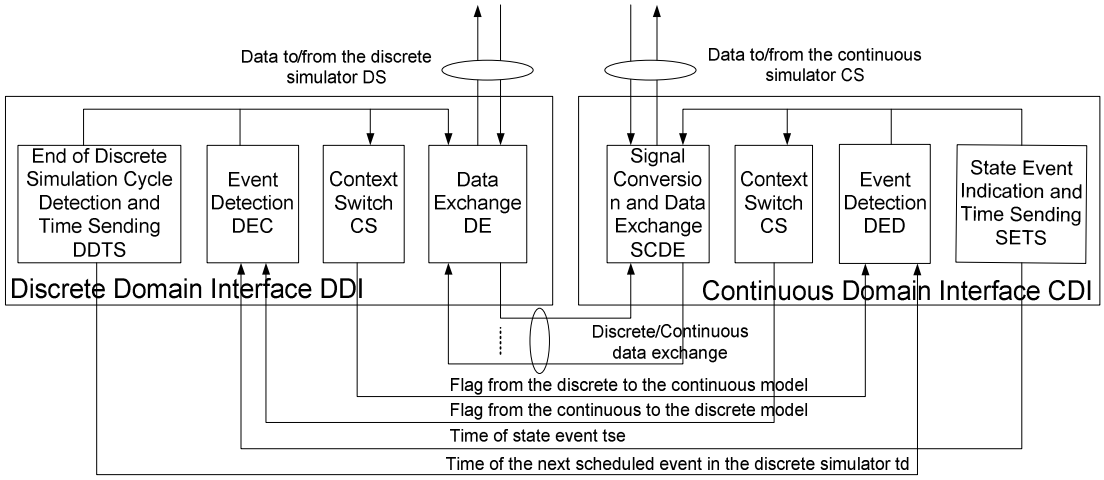


Figure 3.19. Internal architecture of the continuous/discrete simulation interface

The internal architecture is defined as a set of coupled modules that respect the coupled modules DEVS formalism as presented in section 3.2.1:

- $N_{interface}=(X,Y, D, \{M_d | d \in D\}, EIC,EOC,IC)$
- $X = \{(p_d, v_d) | p_d \in InPorts, v_d \in X\, p_d \}$
- $Y = \{(p_d, v_d) | p_d \in OutPorts, v_d \in Y\, p_d \}$
- $X\, p_d$, $Y\, p_d$ = values for input ports, respectively output ports

- $InPorts = P_{in,c} \cup P_{in,d} \cup P_{in,td} \cup P_{in,tse} \cup P_{in,flag}$ where

$P_{in,c}$ – set of ports receiving data from the continuous model; $P_{in,d}$ – set of ports receiving data from the discrete model (via the co-simulation bus);

P_{td} – port receiving the time stamp of the next discrete event

$P_{in,flag}$ – port receiving the command for the context switch

- $OutPorts = P_{out,c} \cup P_{out,d} \cup P_{out,td} \cup P_{out,tse} \cup P_{out,flag}$

$P_{out,flag}$, $P_{out,c}$, $P_{out,d}$ are defined similarly to $P_{in,flag}$, , $P_{in,c}$ and $P_{in,d}$

- D = {"Continuous Domain Interface" (with associated model $N_{interfaceCDI}$), "Discrete Domain Interface" (with associated model $N_{interfaceDDI}$), "co-simulation bus" (with associated model M_{cosim})}

- $M_d = (N_{interfaceCDI}, Mi_{nterfaceDDI}, M_{cosim})$

- EIC = {(($N_{interface}$, "$in_{c,1}$"),($N_{interfaceCDI}$, "$in_{c,1}$")); ...;
  (($N_{interface}$, "$in_{c,n}$"),($N_{interfaceCDI}$, "$in_{c,n}$"));
  (($N_{interface}$, "$in_{d,0}$"),($N_{interfaceDDI}$, "$in_{d,0}$")); ...;
  (($N_{interface}$, "$in_{d,m}$"),($N_{interfaceDDI}$, "$in_{d,m}$"))}

- EOC = {(($N_{interfaceCDI}$, "$out_{c,1}$"),($N_{interface}$, "$out_{c,1}$")); ...;
  (($N_{interfaceCDI}$," $out_{c,p}$"),($N_{interface}$, "$out_{c,p}$"));
  (($N_{interfaceDDI}$,"$out_{d,1}$"),($N_{interface}$, "$out_{d,1}$")); ...;
  (($N_{interfaceDDI}$," $out_{d,q}$"),($N_{interface}$, "$out_{d,q}$"))}

- $IC = \{((N_{interfaceCDI}, op_{CDI}),(M_{cosim}, ip_{cosim}))| N_{interfaceCDI},$

- $M_{cosim} \in D$, $op_{CDI} \in OutPorts_{CDI,}$ $ip_{cosim} \in InPorts_{cosim}\} \cup$

  $\{((N_{interfaceDDI}, op_{DDI}),(M_{cosim}, ip_{cosim}))| N_{interfaceDDI},$

  $M_{cosim} \in D$, $op_{DDI} \in OutPorts_{DDI,}$ $ip_{cosim} \in InPorts_{cosim}\} \cup$

  $\{((M_{cosim}, op_{cosim}),(N_{interfaceCDI}, ip_{CDI}))| N_{interfaceCDI},$

  $M_{cosim} \in D$, $op_{cosim} \in OutPorts_{cosim,}$ $ip_{CDI} \in InPorts_{CDI,}\} \cup$

  $\{((M_{cosim}, op_{cosim}),(N_{interfaceDDI}, ip_{DDI}))| N_{interfaceDDI},$

- $M_{cosim} \in D$, $op_{cosim} \in OutPorts_{cosim,}$ $ip_{DDI} \in InPorts_{DDI,}\}$

We show here the atomic module **co-simulation bus** that can be formally defined as follows:

- $X = \{(p_d, v_d) | p_d \in InPorts, v_d \in X\, p_d\}$

- $Y = \{(p_d, v_d) | p_d \in OutPorts, v_d \in Y\, p_d\}$

- $InPorts = P_{in,c} \cup P_{in,d} \cup P_{in,td} \cup P_{in,tse} \cup P_{in,flag}$

- $OutPorts = P_{out,c} \cup P_{out,d} \cup P_{out,td} \cup P_{out,tse} \cup P_{out,flag}$

where $P_{in,c}$, $P_{in,d}$ $P_{in,td}$, $P_{in,tse}$, $P_{in,flag}$ $P_{out,c}$, $P_{out,d}$, $P_{out,td}$, $P_{out,tse}$, $P_{out,flag}$ as well as $X\, p_d$, $Y\, p_d$ were previously defined.

States triplet S: (phase * σ * job) where:

phase: (“passive”, “active”)

σ : $\Re_0^+$ advance time

job: (“store”, “respond”)

$\boldsymbol{S}$ = {“passive”, “active”} * $\Re_0^+$ * {“store”, “respond”}

$\boldsymbol{\delta_{ext}}$ ((“passive” * σ * job), e, x)=

(“passive”, σ –e, x), if x=0

(“active”, σ –e, job), if x!=0

$\boldsymbol{\delta_{int}}$ (s)= (“active”, σ, job)

λ (“active”, σ, job) = {“store”, “respond” }

$\boldsymbol{t_a}$(phase, σ, job) = σ

The architecture of the discrete domain interface and the continuous domain interface are also formally defined as a set of coupled modules. Formal descriptions for DDI and CDI respect the coupled module DEVS formalism. Each element of the structure follows the concepts presented in Section 3.2.1 and that were applied for the overall continuous/discrete simulation interface.

3.2.6 The Analysis of the Simulation Tools for the Integration in the Co-Simulation Framework

The previous steps that describe the gradual formal definition of the simulation interfaces and the required library elements are independent of the different simulation tools and specification languages used generally for the specification/execution of the continuous and discrete sub-systems. After the analysis of the existing tools we found that Simulink® is an illustrative example of a continuous simulator enabling the control functionalities that were presented in Section 3.1.5. These functionalities can be added in generic library blocks and a given Simulink® model may be prepared for the co-simulation by parameterization and addition of these blocks.

Several discrete simulators present the characteristics detailed in Section 3.1.5. SystemC is an illustrative example. Since it is open source, SystemC enables the addition of the presented functionalities in an efficient way – the scheduler can be modified and adapted for co-simulation. In this way, the co-simulation overhead may be minimized. However, the addition of simulation interfaces is more difficult than in Simulink® because the specifications in SystemC are textual and a code generator is required in order to facilitate the addition of simulation interfaces. The automatic generation of the co-simulation interfaces is very suitable, since their design is time consuming and an important source of errors. The strategy currently used is based on the configuration of the components and their assembly. These components are selected from a co-simulation library.

3.2.7 The Implementation of the Library Elements Specific to Different Simulation Tools

The implementation for the validation of continuous/discrete systems was realized using SystemC for the discrete simulation models and Simulink® for the continuous simulation models.

For Simulink®, the interfaces are functional blocks programmed in C++ using S-Functions [16]. These blocks are manipulated like all other components of the Simulink® library. They contain input/output ports compatible with all model ports that can be connected directly using Simulink® signals. The user starts by dragging the interfaces from the interface components library into the model's window, then parameterizes them, and finally connects them to the inputs and the outputs of his model.

For SystemC, in order to increase the simulation performance, part of the synchronization functionality has been implemented at the scheduler's level, which is a part of the state event management and the end of the discrete cycle detection (detects that there are no more delta cycles at the current time). For the generation of the co-simulation interfaces for SystemC, the implementation of a code generator was necessary. This script has as input user-defined parameters such as sampling periods, number and type of ports, and synchronization ports.

3.3 Conclusion

This chapter presented a generic methodology for the design of efficient continuous/discrete co-simulation tools. The methodology can be divided into two main stages: (1) a generic stage, defining simulation interfaces functionality in a conceptual framework when formal methods for the specification and validation are used, and (2) a stage that provides the implementation of the rigorously defined functionality. Given the importance of the co-simulation interfaces, the methodology concentrates on the co-simulation interfaces, their behavior, as well as two synchronization models that are assured by the interfaces.

The definition of the library elements and the internal architecture of the co-simulation interfaces step represents the foundation for the generation of the co-simulation library and implicitly for the co-simulation interfaces generation. The definition of the operational semantics, and the distribution of the synchronization functionality as well

as their behavior play an important role at the output flow with the behavior of the co-simulation interfaces and the synchronization model. The analysis of the simulation tools for the integration in the co-simulation framework helped choosing the tools that were used for the modeling of the continuous and the discrete simulators while the “implementation of the library elements specific to different simulation tools” constitutes the final implementation of the libraries.

CHAPTER 4. APPLICATION AND EXPERIMENTAL RESULTS

This chapter illustrates the application of the proposed methodology for the design of a co-simulation tool called CODIS[2]. The validation of a real continuous/discrete system, a glycemia-level regulator using CODIS is also proposed.

4.1 CODIS Framework

CODIS is a co-simulation tool designed in our laboratory using the generic methodology proposed in Chapter 3 ([51], [52]). This tool allows continuous/discrete simulation. Simulink® [16] is used for the modeling of the continuous execution model and SystemC [8] for the modeling of the discrete execution model. The co-simulation interfaces that are specific for each domain are automatically generated by selecting components, from a co-simulation library. The inputs in the flow are the continuous model in Simulink® and the discrete model in SystemC which are schematic and textual models, respectively. The output of the flow is the global simulation model (co-simulation model) instance.

For Simulink®, the interfaces can be parameterized starting with their dialog box. Figure 4.1 shows the design flow for the continuous domain model, including the continuous co-simulation interfaces. The user starts by dragging the interfaces from the interface components library into the model's window, then parameterizes them, and finally connects them to the inputs and the outputs of the model. Before the simulation, the functionalities of these blocks are loaded by Simulink® from the .dll libraries. The parameters of the interfaces are the number of input and respectively output ports, their type, and the number of state events.

---

2 This work was realized in collaboration with Ph. D Faouzi Bouchhima from Ecole Polytechnique de Montreal

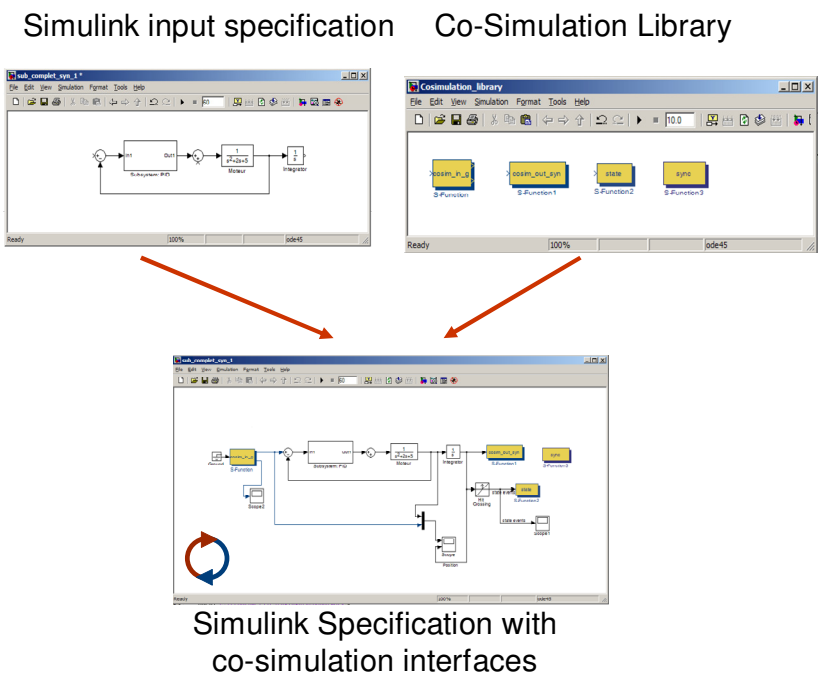


Figure 4.1. Design flow for continuous models

For SystemC, the blocks forming the library are state event management blocks and communication blocks. The interfaces are automatically generated by a script generator that has as input the user-defined parameters. The interface parameters are: the names, the number and the data type of the discrete model inputs ports, and the sampling periods. The tool also generates the function sc_main (or modifies the existing sc_main) that connects the interfaces to the user model. The model is compiled and the link editor calls the library from SystemC and a static library. More details on CODIS can be found in annex 2 ([51], [52]).

The CODIS framework was used to implement a glycemia level regulator that is detailed in the following sections.

4.2 Validation of a Continuous/Discrete System, the Glycemia Level Regulator

The glycemia level regulator is a system enabling a more convenient alternative to the classical therapy for type one diabetes. Type one diabetes, also known as diabetes mellitus (or insulin-dependent) is a permanent condition that takes place when the body's immune system attacks the beta cells that produce insulin in the pancreas and destroy them. The pancreas cannot produce insulin anymore and by consequence the cells cannot use the glucose; a glucose excess builds in the blood. The conventional therapy consists in injections that do not replace the pancreas. A long time supply injection does not answer anymore to the patients needs that can change during the day (because of the alimentation or different effort levels). A new technique is insulin therapy by infusion when a pump infuses insulin or glucose to the patient based on real time values of his glycemia. This application consists in the simulation of a glycemia regulator.

The glycemia system includes two sub-systems, a discrete sub-system, the Control sub-system, for the injection control and a continuous sub-system, the Injection sub-system, for the insulin or glucose injections and the patient model and the glucose assimilation in the blood (as shown in Figure 4.2).

The co-simulation interfaces perform models' adaptation, provide the communication adaptation and the synchronization to accommodate the continuous and the discrete domain. They were generated with respect to the semantics presented in Chapter 2, section 2.2.1, the canonical synchronization model.

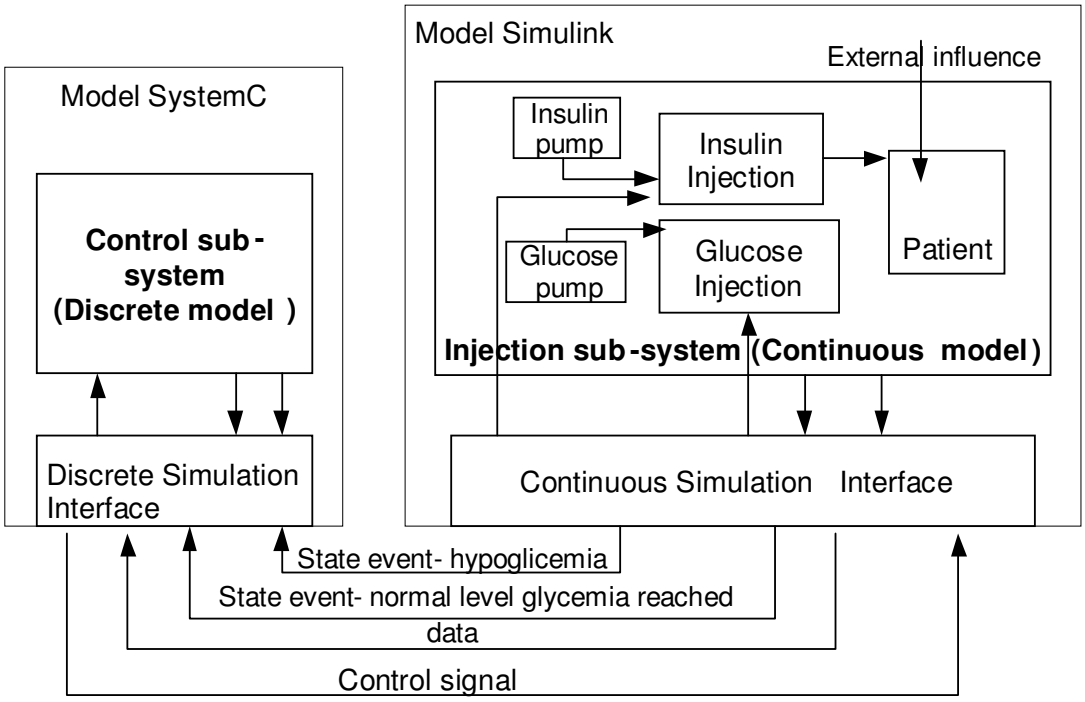


Figure 4.2. The glycemia level regulator system

The patient glycemia level (that is the level of glucose in the blood) is read and compared with the normal level in the "Injection sub-system" and the result is sent to the "Control sub-system". Depending on the value the "Control sub-system" activates either the insulin or the glucose pump. If the level of the glycemia drops under 60mg/dl, this corresponds to the state of hypoglycemia, and the glucose pump will be activated immediately. In the case of this application, two types of state events are generated:

- the state events generated when a normal level of glycemia is reached (120 mg/dl).
- the state events generated when the glycemia drops under a reference value (60 mg/dl)
- hypoglycemia.

Figure 4.3 and Figure 4.4 show the state graph of the Control sub-system and the Injection sub-system, respectively, represented using DEVS. The internal and external transitions illustrate the module's evolution during the simulation.

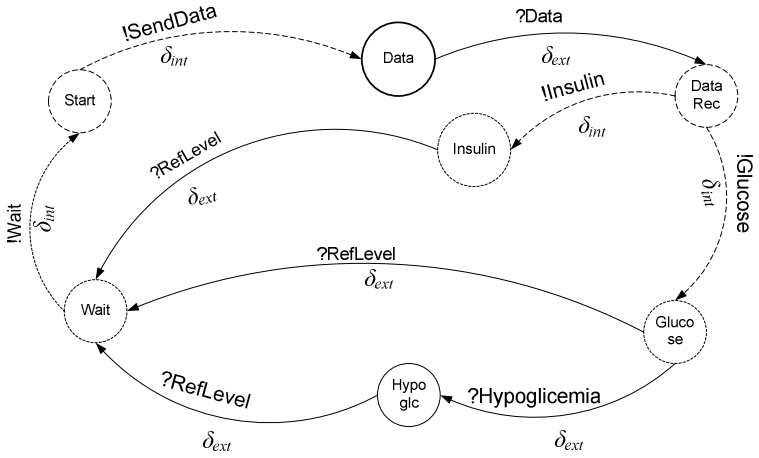


Figure 4.3. State graph of the Control sub-system represented using DEVS

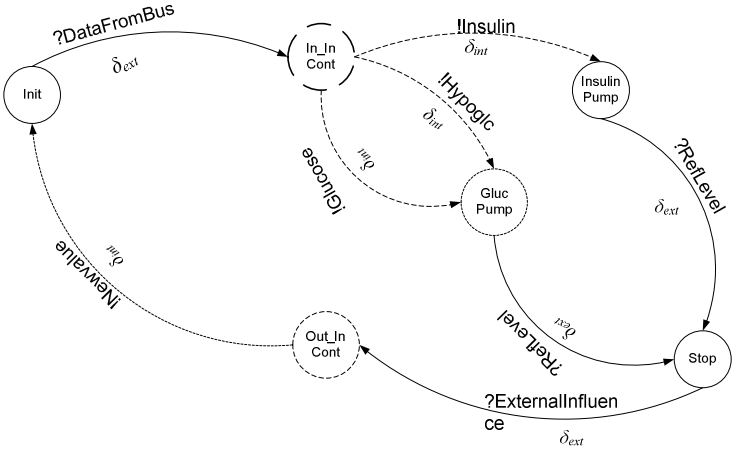


Figure 4.4. State graph of the Injection sub-system represented using DEVS

4.3 Implementation and Results

In the Injection sub-system we have two simulation interfaces for the communication with the discrete sub-system (the Control sub-system), the glucose and the insulin injection sub-modules, the patient model and the block in charge with the state events detection. The state events in this case are generated when a normal level of glycemia is reached or when the glycemia drops under a lower value (hypoglycemia). This module was implemented using Simulink® [16].

The Control sub-system is formed by the two simulation interfaces and a control block that controls if an injection is necessary or not. This module was implemented using SystemC [8].

Figure 4.5(a) and 4.5(b) illustrate the evolution of the patient's insulinemia (units of insulin/dl) during 24 hours monitoring respectively the generation of a state event. The state event is generated at the time 22.2481 when the patient' glycemia reaches the normal level (120mg/dl) (see Figure 4.5(b)). We observe from Figure 4.5(a) that the insulin injection stops at the same time 22.2481, as a consequence of the state event detection. Figure 4.6 shows the messages displayed by the SystemC simulator signalizing the state event detection and the insulin injection.

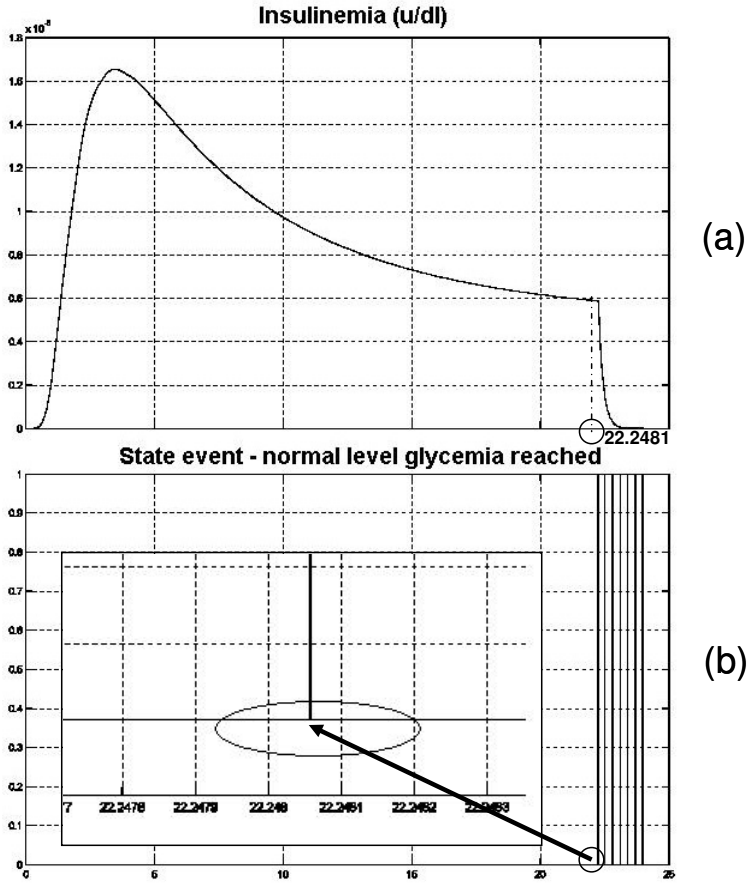


Figure 4.5. Patient's insulinemia (a) and state event generation by CSI (b)

Compared to previous work, a purely discrete control as opposed to a continuous control offers not only a wider range of control features for the pumps but also a more accurate response for events that take place during the injections (like a reference limit of glycemia reached or a hypoglycemia alert followed by glucose injection).

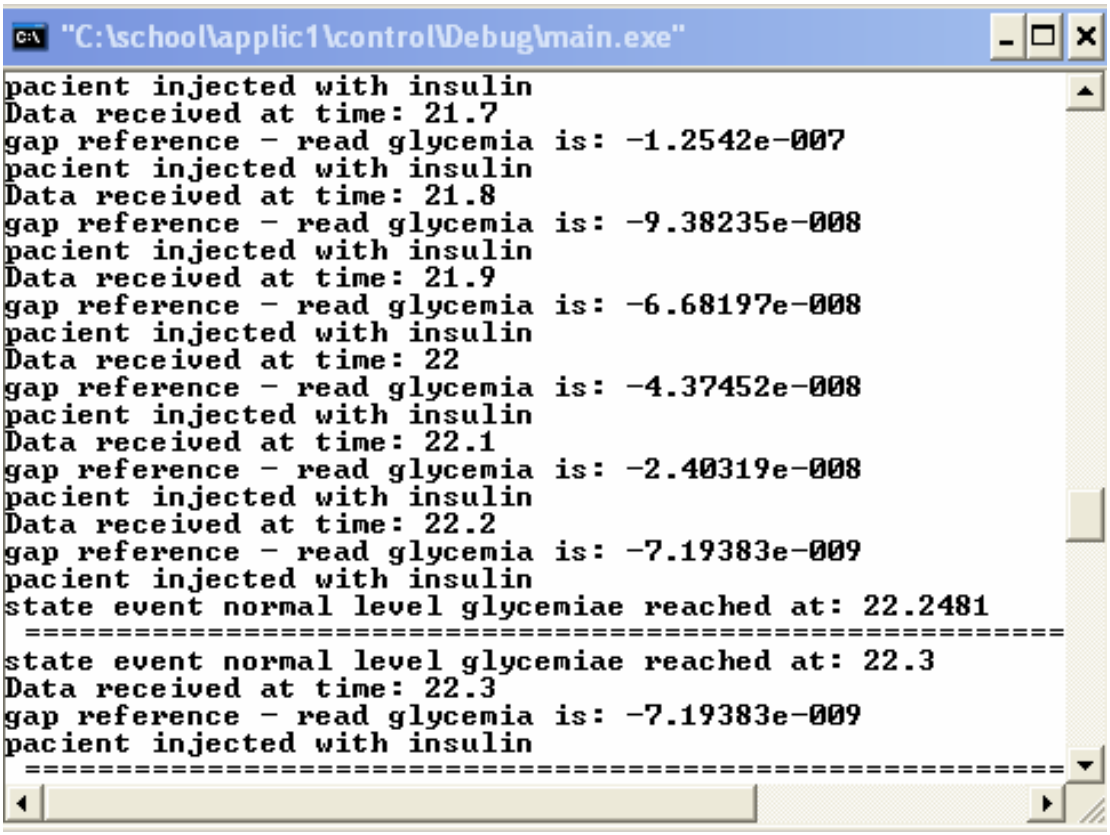


Figure 4.6. State event detection by DSI

4.4 Conclusion

This chapter presented the application of the methodology detailed in Chapter 3. The result of the methodology is a co-simulation framework that allows the modeling and the validation of continuous/discrete heterogeneous systems – CODIS. This framework was validated by implementing a glycemia level regulator. We presented here the results of the co-simulation.

CONCLUSION AND PERSPECTIVES

This research is motivated by the current context in multi-domain embedded systems, where several components from different domains including optical, electrical, mechanical or biological are taken into consideration The unparalleled flexibility of computation has been a key driver in the development of this wide range of products across a broad and diverse spectrum of applications in many industries, but not limited to Automotive, Aerospace, Health Care, Consumer Electronics, and others. These multi-domain heterogeneous systems enable new applications and create new markets. Continuous-time and discrete-event models are at the core of the design of multi-domain systems.

In this last part of this document we present the summary of the thesis and the directions for future research.

1. Summary of the Thesis

Chapter 1 presented a review of the existing works in the modeling and validation of continuous/discrete heterogeneous systems. These works were classified in two extensive categories: simulation-based and formal representation-based approaches. The first category can also be divided into two separate classes:

- A homogeneous approach that consists of the use of only one language for the global specification of the behavior of the system;
- A heterogeneous approach that consists of the use of different languages that are specific for the different sub-systems domains, therefore, they conserve the domain specific descriptions of the modules and the models are simulated in parallel.

The formal representation-based category consists of the representation of the heterogeneous system in a pragmatical mathematical language. The model's validation

is realized by formal verification using different techniques such as theorem proving or model checking.

Chapter 2 defined the global execution model of continuous/discrete heterogeneous systems as well as the execution models for each simulator: the discrete execution model and the continuous execution model. An execution model can be viewed as the interpretation of a computation model. This chapter also details two synchronization models that can be taken into consideration when co-simulating the C/D systems:

- Canonical synchronization model – when the continuous simulator advances before the discrete simulator. In this case, if a state event is generated by the continuous simulator, the need of rollback (and also supplementary resources) is eliminated.
- Rollback-based synchronization model – when the discrete simulator advances before the continuous simulator. In this case, if the continuous simulator generates a state event, the discrete simulator will backtrack to the previous known stable state (light rollback).

When an unpredictable event is generated by the continuous simulator, the discrete simulator has to update the events in its events queue. This chapter also presents the update events schema for both synchronization models.

Chapter 3 proposed a generic methodology for the development of C/D co-simulation tools that is independent of the languages used to implement the two simulators (i.e. Simulink® or Spice for the continuous simulator and SystemC, SystemVerilog, VHDL for discrete simulator). The methodology is divided into two stages: a generic stage where the model is gradually refined from its operational semantics (that gives a pragmatic description) to the definition of the internal architecture of the co-simulation interfaces and the library elements. The second stage is the implementation stage where the simulation tools are analyzed, the library elements are implemented and the model is validated. The methodology was demonstrated using DEVS formalism, timed automata

and UPPAAL for the generic stage and SystemC and Simulink® for the discrete, respective continuous models for the implementation stage.

In Chapter 4 we presented the application of the methodology for the definition of a framework for the modeling and simulation of C/D heterogeneous systems – CODIS. This framework was used for several concrete applications such as: control systems, continuous systems. In this thesis we present a glycemia regulator implemented using SystemC and Simulink®. The results of the co-simulation are presented in Chapter 4.

A summary of the major contributions is listed below:

- The analysis of the execution models and the synchronization models for continuous/discrete systems.
- The definition of a generic methodology for the efficient design of continuous/discrete co-simulation tools. Before the implementation stage, the methodology suggests several steps enabling the gradual formal definition of the simulation interfaces functionality and internal architecture:
    - The definition of the operational semantics for a continuous/discrete synchronization model.
    - The formal representation of the behavior of continuous/discrete co-simulation interfaces, with respect to a synchronization model.
    - The formal verification of the behavior of continuous/discrete interfaces.
    - The description of the internal architecture of the continuous/discrete co-simulation interfaces.
- The application of the methodology – the development of a co-simulation framework – CODIS and the implementation of a glycemia level regulator. Parts of the methodology were also used for the formalization, the modeling and the verification of components of an Optical Network on Chip (ONoC). This work is detailed in annex 1.

2. Directions for Future Research

This thesis makes strides toward the development of a generic methodology for the design of continuous/discrete heterogeneous systems co-simulation tools and opens new directions important for the researchers that work in system level simulation. The methodology proposed here allows for new developments in the automatic generation of the co-simulation interfaces for continuous/discrete heterogeneous systems. A new research direction opened by this work is the formal verification of the composition of the elements of the library in order to create an interface. Another area that can be covered is the analysis of the continuous and discrete models to be integrated in order to verify their compatibility in terms of inputs, outputs, abstraction levels.

This work can be continued with modeling and simulation of C/D heterogeneous systems at different levels of abstraction and the integration of the rollback-based synchronization model in the co-simulation framework. New domain specific simulation tools (such as SystemVerilog for the discrete domain) can be integrated in order to validate the genericity of the methodology. Some work might also be conduct for performances analysis and methodology optimization.

Another area in which the presented work can be used is ONoC modeling and validation. The next step in this direction is the integration of the passive and the active optical devices and IC in order to realize the global execution model of the ONoC. Moreover, interconnects play a significant role for MPSoC design. Integrated optical interconnects are interesting alternative to traditional interconnects because they overcome current limitations like bandwidth, contention and latency. The access to physical prototyping of ONoC is challenging therefore high-level modeling and validation are mandatory. On a long term the methodology proposed here can be adapted for the modeling and validation of MPSoC integrating ONoC.

REFERENCES

[1] International Technology Roadmap for Semiconductor Design (ITRS) [Online]. Available: http://public.itrs.net/.

[2] S. Romitti, C. Santoni, P. Francois, P. “A Design Methodology and a Prototyping Tool Dedicated to Adaptive Interface Generation”, in *Proceedings of the 3rd ERCIM Workshop on "User Interfaces for* All", Obernai, 1997.

[3] M. Keating, P., Bricaud, “Reuse methodology manual for system-on-a-chip designs.” Kluwer Academic Publishers, Boston, 2002.

[4] Y. Monsef, *Modelisation et simulation des systèmes complexes.* Lavoisier tec et doc, Paris, 1996.

[5] A. A. Jerraya, *Conception de haut niveau des systèmes monopuces.* Hermès Science Publications, Paris, 2002.

[6] VHDL [Online], Available: http://www.vhdl.org/vhdl-200x/.

[7] Verilog [Online]. Available: http://www.verilog.com/IEEEVerilog.html.

[8] SystemC 2.0.1 Language Reference Manual Revision 1.0, 2003, [Online] Available: http://www.SystemC.org.

[9] A. Doboli, R. Vemuri, “Behavioral modeling for high-level synthesis of analog and mixed-signal systems from VHDL-AMS”, in *IEEE Transactions On Computer Aided Design of Integrated Circuits and Systems* , vol.22, no. 11, Nov. 2003.

[10] IEEE Standard VHDL Analog and Mixed-Signal Extensions, IEEE Std 1076.1-1999, 23 Dec. 1999.

[11] J.-J. Charlot, N. Milet-Lewis, T. Zimmer, H. Levi. “VHDL-AMS for mixed technology and mixed signal, an overview”, in *MED’01*, Dubrovnik, Croatia, 27-29 June 2001.

[12] P. Frey, D. O'Riordan, “Verilog-AMS: Mixed-signal simulation and cross domain connect modules”, in *Proceedings of BMAS IEEE/ACM International Workshop*, 2000.

[13] Verilog-AMS LRM version 2.3,[Online]. Available: http://www.accellera.org/activities/verilog-ams/VAMS-LRM-2-3.pdf.

[14] A. Vachoux, C. Grimm, K. Einwich, “SystemC-AMS requirements, design objectives and rationale”, *Proceedings of the DATE (DATE’03),* 2003.

[15] A. Vachoux, C. Grimm, K. Einwich, “Analog and mixed signal modeling with SystemC-AMS”, Circuits and Systems, *Proceedings of ISCAS'03*, 2003.

[16] Matlab-Simulink, [Online]. Available http://www.mathworks.com.

[17] D.H. Patel, S. K. Shukla: “SystemC Kernel – Extensions for heterogeneous System Modeling” Kluwer Academic Publishers, Boston. 2004.

[18] Ptolemy, University of California, Berkeley, [Online]. Available: www.ptolemy.eecs.berkeley.edu/ ptolemyII/.

[19] T. Kurzweg, S. Levitan, P. Marchand, et al “Modeling an Simulating Optical MEMS Using Chatoyant”, *Design, Test, and Microfabrication of MEMS/MOEMS*, Paris, 1999.

[20] S. Levitan, J. Martinez, T. Kurzveg, P. Marchand, D. Chiarulli, “Multi technology system-level simulation” *Design Test Integration and Packaging of MEMS/MOEMS (DTIP 2000)*, May 2000.

[21] G. Nicolescu, Y. Sungjoo, A. Bouchhima, A. A. Jerraya, “Validation in a component-based design flow for multicore SoCs” in *Proceedings of the 15th ISSS (ISSS’02)* Kyoto, Japan 2002.

[22] G. Nicolescu, S. Martinez, L. Kriaa, W. Youssef, S. Yoo, B. Charlot, A.A. Jerraya, "Application of Multi-domain and Multi-language Cosimulation to an Optical MEM Switch Design ", *ASP-DAC 2002,* Bangalore, India, January 2002.

[23] E. A. Lee, and H. Zheng, "Operational Semantics of Hybrid Systems" in *Hybrid Systems: Computation and Control: 8th International Workshop, HSCC*, 2005.

[24] G. Bosman, "A survey of Co-design Ideas and Methodologies", thesis report, 2003, [Online] Available: http://www.guusbosman.nl/downloads/thesis20030225.pdf.

[25] E.A. Lee, and A.L. Sangiovanni-Vincentelli: "Comparing Models of Computation", In: *Proceedings of the ICCAD'96*, IEEE Computer Society, 1996.

[26] A. Jantsch, *Modeling Embedded Systems and SoC's – Concurrency and time in Models of Computation*, Morgan Kaufmann Publishers, San Francisco, 2004.

[27] A. Jantsch, and I. Sander, "Models of Computation and Languages for Embedded System Design", IEE Proc.-Comput. Digit. Tech., vol. 152, 2005.

[28] B. P. Zeigler, *Theory of modeling and simulation*, Wiley Interscience, 1976.

[29] B. P. Zeigler, H. Praehofer and T. G. Kim, *Modeling and Simulation – Integrating Discrete Event and Continuous Complex dynamic Systems*, Academic Press, San Diego, 2000.

[30] M. D'Abreu and G. Wainer, "M/CD++ : modeling continuous systems using Modelica and DEVS", in *Proc. of the IEEE Int. Symposium of MASCOTS'05*, 2005, pp 229 – 238.

[31] Y. J. Kim, J. H. Kim, and T. G. Kim, "Heterogeneous simulation Framework using DEVS-BUS", in: *Simulation, the Society for Modeling and Simulation International,* vol. 79, 2003, pp. 3-18.

[32] G. Wainer: "Modeling and Simulation of Complex Systems with Cell-DEVS". *Winter Simulation Conference* 2004, pp 49-60.

[33] J-S Bolduc, H. Vangheluwe "The modelling and simulation package PythonDEVS for classical hierarchical" DEVS. MSDL technical report MSDL-TR-2001-01, McGill University, June 2001.

[34] T. G. KIM, DEVSim++ User's Manual, SMSLab, Dept. of EECS, KAIST, 1994, [Online]. Available: http://smslab.kaist.ac.kr .

[35] V. Madisetti, J. Walrand, and D. Messerschmitt, "WOLF: A rollback algorithm for optimistic distributed simulation systems" in: *Proc of the 1988 Winter Simulation Conference*, 1988 pp. 296-305.

[36] T. H. Feng and E. A. Lee, " Incremental checkpointing with application to distributed discrete event simulation" in: *Proc of the 2006 Winter Simulation Conference*,2006.

[37] J. Fleischmann et al. "Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators", in *Parallel and Distributed Simulation*, 1995.

[38] C.G. Cassandras, *Discrete event systems: Modeling and performance analysis*. Richard Irwin, New York, 1993.

[39] SystemVerilog, [Online] Available: http://www.systemverilog.org/.

[40] G. K. Gupta, R. Sacks-Davis, P. E. Tischer, "A review of recent developments in solving ODES", in *Proceedings of the CSUR*, vol. 17, issue 1, 1985.

[41] SPICE, [Online]. Available: http://bwrc.eecs.berkeley.edu/Classes/icbook/SPICE/.

[42] F.E. Cellier, "Combined Continuous/Discrete System Simulation Languages - Usefulness, Experiences and Future Development", *Methodology in Systems Modeling and Simulation,* North-Holland, Amsterdam, 1979, pp.201-220.

[43] H. R. Ghasemi, "An effective VHDL-AMS simulation algorithm with event", *International Conference on VLSI Design*, 2005.

[44] F. Wang, "Formal verification of times systems: a survey and perspective" , in *Proc. of the IEEE*, vol. 92, 2004, pp 1283-1305.

[45] J-F. Monin*, Understanding Formal Methods*. Springer, 2003.

[46] R. Alur and D. Dill, “Automata for modeling real-time systems”, in *Proc. 17-th Int. Colloquium on Automata, Languages and Programming*, vol. 443, 1990, pp. 322-335 .

[47] J. Bengtsson, and W. Yi, “Timed automata: Semantics, Algorithms and Tools”, Uppsala University. Denmark,1996.

[48] G. Behrmann, A. David and K. Larsen, “A Tutorial on UPPAAL”, *Real-Time Systems Symposium,* Miami, 2005.

[49] N. Giambiasi, J-L. Paillet, F. Chane, ”From timed automata to DEVS”, in *Proc. of the 2003 Winter Simulation Conference*, 2003.

[50] S. Edwards, L. Lavagno, E. Lee and A.L. Sangiovanni-Vincentelli, “Design of Embedded Systems: Formal Models, Validation, and Synthesis” in *Proc. of the IEEE*, vol. 85, 1997, pp. 366-390.

[51] F. Bouchhima, G. Nicolescu, M. Aboulhamid and M. Abid. “Discrete–Continuous Simulation Model for Accurate Validation in Component-Based Heterogeneous SoC Design”, *Rapid Systems prototyping*, 2005, pp 181-187.

[52] F. Bouchhima, G. Nicolescu, M. Aboulhamid and M. Abid, “Generic discrete–continuous simulation model for accurate validation in heterogeneous systems design”, in *Microelectronics Journal*, vol. 38, 2007, pp 805-815.

[53] M. Brière, L. Gheorghe, G. Nicolescu, I. O’Connor, G. Wainer. “Towards the high level design of optical networks on chip. Formalization of opto-electrical interfaces”, in *Proc. of the IEEE ICECS*, Morocco, 2007.

[54] M. Briere, L. Carrel, T. Michalke, F. Mieyeville, I. O’Connor and F. Gaffiot. “Design and behavioral modeling tools for optical network-on-chip”, in *Proc. of the DATE*, 2004.

[55] M. Briere, E. Drouard, F. Mieyeville, D. Navarro, I. O'Connor and F. Gaffiot.: Heterogeneous Modeling of an Optical Network-on-Chip with SystemC in *Proc.of the Rapid System Prototyping (RSP)*, 2005.

[56] L. Benini, G. Di Micheli, "Networks on chips: a new SoC paradigm", *Computer*, vol. 35, no.1, 2002, pp. 70-77.

[57] I. O'Connor, "Optical solutions for system-level interconnect", in *Proc. of the International workshop on SLIP*, Paris France, 2004.

[58] Celoxica available online at http://www.celoxica.com/methodology .

[59] A. Kazmierczak, et al., "Design, Simulation and Characterization of a Passive Optical Add-Drop Filter in Silicon-On-Insulator Technology", in *IEEE Photonic Tech. Lett*., vol. 17, 2005, pp. 1447 – 1449.

[60] M. Kobrinsky, M. et al. "On-chip optical interconnects", in *Intel Technology Journal*, vol. 8, no. 2, 2004, pp. 129-142.

ANNEX 1 – COMPLEMENTARY RESULTS
OPTICAL NETWORK ON CHIP MODELING AND VALIDATION

This annex presents: the formalization of optical-electrical interfaces using DEVS[3], the formalization of basic elements of an optical network on chip using DEVS[3] and the modeling and the formal verification for the global validation of the behavior of a passive optical network on chip using timed automata ([53], [54] and [55]).

1 Optical Networks on Chip

Many of the modern Systems-on-Chip integrate a high density of heterogeneous components such as different processors, a wide range of hardware components, as well as complex interconnects that use different communication protocols. On-chip physical interconnections represent a limiting factor for performance and energy consumption. Energy and device reliability impose small logic swings and power supplies. Moreover, the growth of the number of components that are integrated on-chip increases the impact of the deep sub-micron effects (ex. electrical noise due to crosstalk, electromagnetic interference can produce data errors). By consequence, transmitting data on wires may be in some cases unreliable and nondeterministic [56]. New interconnect challenges are added when moving to 65nm and beyond: interconnect delay becomes larger than gate delay and the interconnect area becomes much larger than the gate area [56]. Designers also face deep sub-micron effects like voltage isolation and wave reflection. Optical Networks on Chip (ONoC) are promising because of their scalability, simplicity and low real estate (0.00425 mm^2 for passive network) [57]. However, the access to physical prototyping for multi-technology SoCs is a major challenge because of its significant cost and the difficulty to influence standard processes. Modeling and simulation become

3 This work was realized in collaboration with Ph. D Mathieu Brière et Prof. Dr. Ian O'Connor, Ecole Centrale de Lyon, France

necessary alternatives in the design space exploration for these systems. Today, in many application designs the most costly task in terms of time and human resources is the design verification. Formal methodologies emerge as a more structured verification approach [1]. This implies that the design model is more thorough checked and more cases are taken into consideration.

The methodology presented in Chapter 3 of this thesis can help the designer to achieve the complex design of these systems, and thus reduce the design process.

The integrated optical communication system studied in this work, also called Optical Network on Chip (ONoC) [57] is composed of three types of blocks: *i)* a transmitter interface circuits (for the electro-optical conversion) *ii)* a passive integrated photonic routing structure (named λ- router) and iii) a receiver interface circuit (for the opto-electrical conversion).

Figure 4 presents an overview of this ONoC plugging initiators and targets (also called cores). The ONoC is a heterogeneous structure that can be represented as a combination of passive and active optical devices as well as mixed analog/digital integrated circuits.

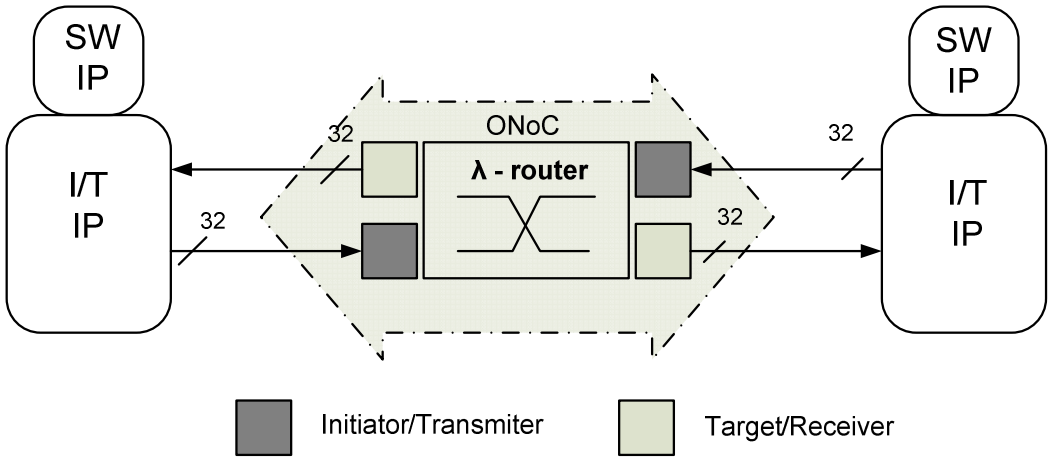


Figure 4. ONoC overview (I=Initiator, T=Target)

2 Formalization of Optical-Electrical Interfaces

This section presents the DEVS formalism applied to optical-electrical interfaces. We take into consideration only the functional conversion interfaces in order to prove the DEVS efficiency for the optical formalism. This methodology can be then applied to easily design more complex systems using DEVS coupled models.

2.1 Transmitter Architecture

Each SoC core (initiator and target) requires a transmitter block which enables the electro-optical conversion (as shown in Figure 5). This block is mainly composed of a laser to emit light at a given wavelength and optical power, and its driver for the modulation and polarization.

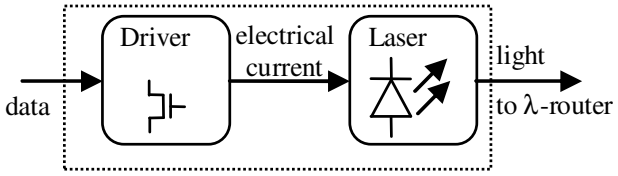


Figure 5. Optical transmitter architecture

Figure 6 shows the optical transmitter architecture with respect for the DEVS formalism, including the internal and external events with the Is/Os.

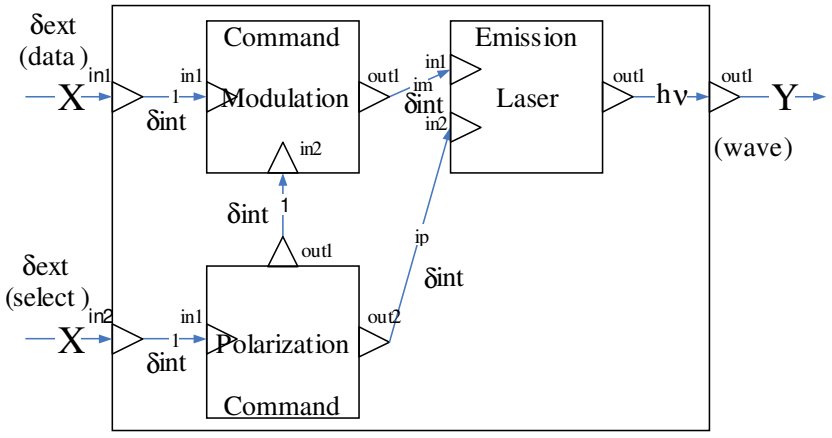


Figure 6. Optical transmitter architecture with DEVS notations

Next equations give the formal description of the optical transmitter (electro-optical conversion) using DEVS:

$DEVS_{TX} = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta)$

(6)

with :

inputs: $InPorts = \{\text{'data'}, \text{'select'}\}$

input set: $X = \{(p,v) | p \in InPorts, v \in X_p\}$ with $X_p = \{data\_to\_send\} \setminus \{activation\}$

and,

outputs: $OutPorts = \{\text{'wave'}\}$

output set: $Y = \{(p,v) | p \in OutPorts, v \in Y_p\}$ with $Y_p = wave\_value \in \{wavelength, power\}$

The *states* are: $S = \{\text{'idle'}, \text{'conversion'}\}$

(7)

The *internal events* are:

$\delta_{int}(phase, \sigma, local\_inport, local\_value, inport, value): S \rightarrow S$

$= (\text{'modulation'}, \sigma, p, v, latency\_mod)$

if $phase = \text{'conversion'}$ and $p = modulation\_port$ and $v = \{data\_to\_send\}$

$= (\text{'polarization'}, \sigma, p, v, latency\_pol)$

if $phase = \text{'conversion'}$ and $p = polarization\_port$ and $v = \{active\}$

or if $phase = \text{'idle'}$ and $p = polarization\_port$ and $v = \{no\_active\}$

$= (\text{'emission'}, \sigma, p, v, latency\_laser)$

if $phase = \text{'conversion'}$ and $p = laser\_port$ and $v = wave\_value$ (with *power* proportional with the modulation current I_m and the polarization current I_p of the laser driver).

$= (\text{'idle'}, \sigma, p, v)$ else.

The *external events* are:

$\delta_{ext}(phase, \sigma, e, x): Q \times X \rightarrow S$

$= (\text{'idle'}, e, p, v)$

if $phase = \text{'idle'}$ and $p = activation$ and $v = off$

$= (\text{'busy\_active'}, process\_time, p, v)$

if $phase = \text{'conversion'}$ and $p = activation$ and $v = on$

$= (\text{'busy\_send'}, process\_time, p, v)$

if $phase = \text{'conversion'}$ and $p = data$ and $v = data\_to\_send$

with $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$

The *output functions* are:

$\lambda(phase, \sigma, local\_inport, wave\_value, wave): S \rightarrow Y$

$= (out, wave\_value)$

if $phase = \text{'conversion'}$ and $local\_inport = laser\_port$

$= (out, 0 \cdot exp(0))$

if $phase = \text{'idle'}$

The *state advancing time* is:

$t_a(phase, \sigma): S \rightarrow \Re^+_{0,\infty} = \sigma = latency \mid time\_next\_data$

(8)

with $latency = latency\_mod \mid latency\_pol \mid latency\_laser$

The transmitter's behavior (as seen in (7)) is characterized by two states: *idle* (no conversion) and *conversion* (data is sent through the interface). There are 4 internal events: *modulation* (to modulate the laser with the data to convert), *polarization* (to polarize the laser), *light* (for the light emission at a given optical power and wavelength) and *idle* (no light emission); and 3 external events: *idle*, (no conversion) *selection* (conversion activation) and *data* (data to convert). The state advancing time shown in (8) is mainly composed of latencies extracted from physical design (IC) or datasheet (laser).

2.2 Receiver Architecture

Similar to the transmitter block, each SoC core requires a receiver block which enables the opto-electronic conversion (as shown Figure 7). This block is mainly composed of a

photodiode (conversion of flow of photons into photocurrent), a TransImpedance Amplifier (TIA), a decision circuit (digital signal regeneration).

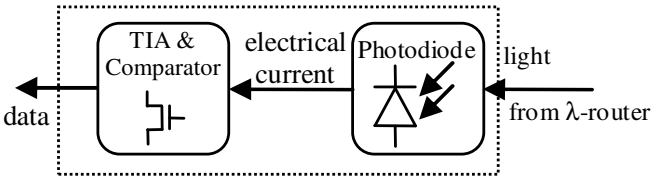


Figure 7. Optical receiver architecture

Figure 8 shows an optical receiver architecture, including the internal and external events with the Is/Os, with respect to the DEVS formalism.

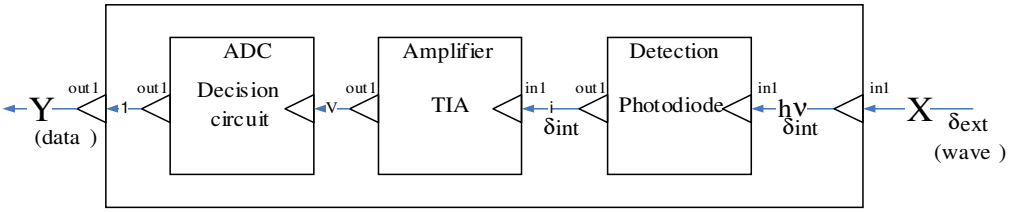


Figure 8. Optical receiver architecture with DEVS notations

$DEVS_{RX} = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta)$ (9)

with:

inputs: $InPorts = \{\text{'wave'}\}$

input set: $X = \{(p,v) | p \in InPorts, v \in X_p\}$ with $Xp = wave\_value \in \{wavelength, power\}$

and,

output: $OutPorts = \{\text{'data'}\}$

output set: $Y = \{(p,v) | p \in OutPorts, v \in Y_p\}$

with $Y_p = \{data\_to\_receive\}$

The *states* are: $S = \{\text{'idle'}, \text{'conversion'}\}$ (10)

The *internal events* are:

$\delta_{int}(phase, \sigma, local\_inport, local\_value, inport, value): S \rightarrow S$

$= (\text{'detection'}, \sigma, p, v, latancy\_pdiode)$

if $phase = \text{'conversion'}$ and $p = pdiode\_port$ and $v = wave\_value$

$= (\text{'amplifier'}, \sigma, p, v, latency\_TIA)$

if $phase = \text{'conversion'}$ and $p = TIA\_port$ and $v = photocurrent$

$= (\text{'ADC'}, \sigma, p, v, latency\_ADC)$

if $phase = \text{'conversion'}$ and $p = ADC\_port$ and $v = photocurrent \cdot gain$

$= (\text{'idle'}, \sigma, p, v)$ else.

The *external events* are:

$\delta_{ext}(phase, \sigma, e, x): Q \times X \rightarrow S$

$= (\text{'idle'}, e, p, v)$

if $phase = \text{'idle'}$ and $p = wave$ and $v = 0 \cdot exp(0)$

$= (\text{'busy\_receive'}, process\_time, p, v)$

if $phase = \text{'conversion'}$ and $p = wave$ and $v = wave\_value$

with $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$

The *output functions* are:

$\lambda(phase, \sigma, local\_inport): S \rightarrow Y$

$= (out, data\_to\_receive)$

if $phase = \text{'conversion'}$ and $data\_to\_receive = bit\_value$ and $local\_inport = ADC\_port$

$= (out, \text{'X'})$ if $phase = \text{'idle'}$

The *state advancing time* is:

$$t_a(phase, \sigma): S \rightarrow \Re^+_{0,\infty} = \sigma = latency \quad (11)$$

with $latency = latency\_pdiode \mid latency\_TIA \mid latency\_ADC$

The two states (that characterize the receiver's behavior) were taken into as shown in (10): *idle* (there is no conversion) and *conversion* (data is detected through the interface). However, the behavior of the receiver is easier than the receiver. There are 4 internal events: *photoconversion* (for the light conversion in photocurrent), *amplify* (for the amplification of the the current and the conversion in voltage), *CAN* (for the analog-

to-digital conversion) and *idle* (no light to detect); and 2 external events: *idle*, (no conversion) and *data* (light to convert). The state advancing time shown in (11) is mainly composed of latencies extracted from physical design (IC) or datasheet (photodiode).

2.3 Passive Photonic Devices

The λ - router is a passive optical network (as shown in Figure 9(a)) composed of 4-port optical switches based on add-drop filters (as shown in Figure 9(b)) designed to route data through SoC components ([53], [54], [55]). These add-drop filters operate in a similar way to classical electronic switches. An optical filter is characterized by a specific wavelength, called resonant wavelength (λ_i in the Figure 9) depending on filter geometry and material. Figure 9(a) presents an example of a $N \times N$ λ-router architecture (each grey square representing an add-drop filter) and a physical architecture example of the filter is shown Figure 9(b). The add-drop is bidirectional and compact devices have been demonstrated in CMOS compatible Silicon on Insulator (SOI) technology (Si/SiO_2 structures accept 1.3-1.55 μm wavelength) [59].

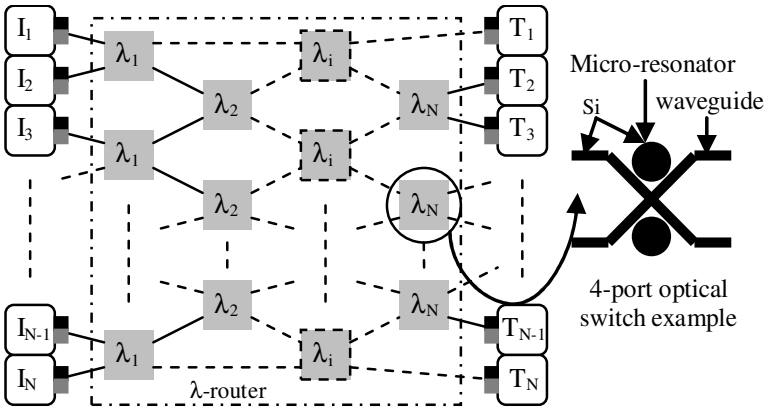


Figure 9. *N x N λ-router* architecture (a), 4-port optical switch architecture example (b)

As illustrated in Figure 10, there are three possible switch states depending on the input signal:

- Straight state 10(a) occurs when specific wavelengths, called resonant wavelengths (λ_i, depending on micro-resonator geometry and material) are injected in the filter and are routed through the micro-resonator.

- Diagonal state 10(b) occurs when other wavelengths (λ_j) are injected in the filter and are not routed through the micro-resonator.

- Cumulative state 10(c) occurs when signals of both resonant and non-resonant wavelengths (λ_i and λ_j) are injected into the filter using the WDM technique[4] and are either routed or not routed through the micro-resonator. Because of this property and the fact that the four add-drop ports can be used simultaneously, a contention-free network can be built.

- Possible exploitation of the optical switch is shown in 10(d). This example shows both unidirectional and bidirectional behaviors (several wavelengths simultaneously injected in opposite way).

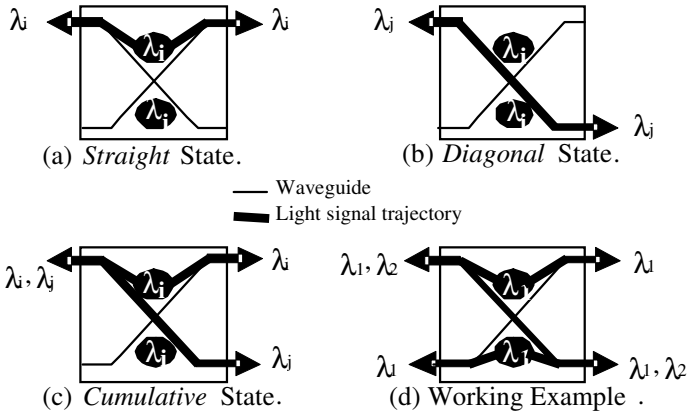


Figure 10. Functional states of a 4-port optical switch

[4] Wavelength Division Multiplexing. Several signals at different wavelengths can be injected into the same waveguide.

The main advantage of this architecture is its high scalability.

Table 2. *4X4 λ-router* truth table

I/T	T_1	T_2	T_3	T_4
I_1	λ_2	λ_3	λ_1	λ_4
I_2	λ_3	λ_4	λ_2	λ_1
I_3	λ_1	λ_2	λ_4	λ_3
I_4	λ_4	λ_1	λ_3	λ_2

Currently, up to 32 cores (16 initiators and 16 targets) can be plugged onto an ONoC, where the limit is due to the lithographical tolerance in add-drop manufacturing. In a λ-router, only one physical path associated with one wavelength exists between an initiator I_i and a target T_j. The broadcast is also possible with this architecture.

In Table 2 we give the truth table for a 4×4 network. For example, if I_2 communicates with T_4, data must use the wavelength λ_1 to be sent through the λ-router. At the same time I_1 can communicate with T_1 using the wavelength λ_2. These optical switches and λ-router have been manufactured and tested. The observed network routing corresponds to theory [60].

This section presents two basic passive photonic devices composing a λ – router: a simple point to point connection and a basic 4-port optical switch. We also detail a 4×4 λ-router using these elementary blocks.

2.3.1 Point to Point Optical Connection

Figure 11 shows a point to point bidirectional optical connection with respect to the DEVS formalism notations. A point to point connection can be a straight optical waveguide or a curve for example.

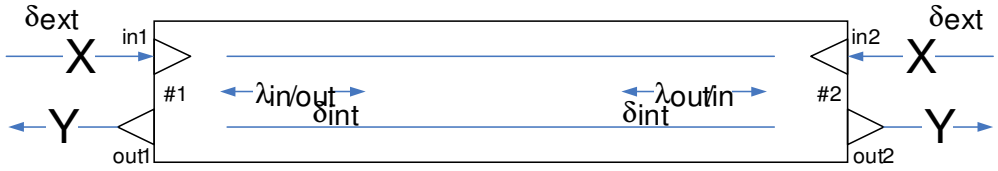


Figure 11. Point to point bidirectional optical connection with DEVS

$DEVS_{P2P} = (X, Y, S, \delta_{int}, \lambda, ta)$ (12)

with:

inputs: $InPorts = \{'in1', 'in2'\}$

input set: $X = \{(p,v) | p \in InPorts, v \in X_p\}$

with: $X_p = wave\_value \in \{wavelength, power\}$

and,

output: $OutPorts = \{'out1', 'out2'\}$

output set: $Y = \{(p,v) | p \in OutPorts, v \in Y_p\}$

with: $Y_p = wave\_value \in \{wavelength, power\}$

The *states* are: $S = \{'idle', 'communication'\}$ (13)

The *internal events* are:

$\delta_{int}(phase, \sigma, inport, wave\_value): S \rightarrow S$

$= ('busy', \sigma, p, v)$

if $phase = 'communication'$ and $p \in InPorts$ and $v \in X_p$

$= ('idle', \sigma, p, v)$ else.

The *external events* are:

$\delta_{ext}(phase, \sigma, e, x): Q \times X \rightarrow S$

$= ('idle', e, p, v)$

if $phase = 'idle'$ and $p = wave$ and $v = 0 \cdot exp(0)$

$= ('in\_light', process\_time, p, v)$

if $phase = 'communication'$ and $p = wave$ and $v = wave\_value \cdot P2Pdefects$

with $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$

The *output functions* are:

$\lambda(phase, \sigma, inport, wave\_value):S \rightarrow Y$

$= (out2, wave\_value \cdot P2Pdefects)$

if $phase$ = *'communication'* and $inport = in1$

$= (out1, wave\_value \cdot P2Pdefects)$

if $phase$ = *'communication'* and $inport = in2$

The *state advancing time* is:

$$t_a(\sigma):S \rightarrow \Re^{+}_{0,\infty} = \sigma = bit\_propagation\_time \quad (14)$$

Two states characterize the point to point connection behavior, as seen in (13): *idle* (no conversion) and *communication* (light is transported through the optical waveguide). There are 2 internal events: *busy* (light is present), *idle* (no light through the waveguide); and 2 external events: *idle* (no light) and *in_light* (light in one of the inputs). The state advancing time shown in (14) is due to the light transport in a waveguide depending on its length and its manufacture materials. This description must take into account the attenuation in the point to point connection due to its defects (*P2Pdefects*). These defects attenuate the optical power value at the outputs.

2.3.2 Four Port Optical Switch

Figure 12 shows a point to point bidirectional optical connection with respect to the DEVS formalism notations [53], [54], [55].

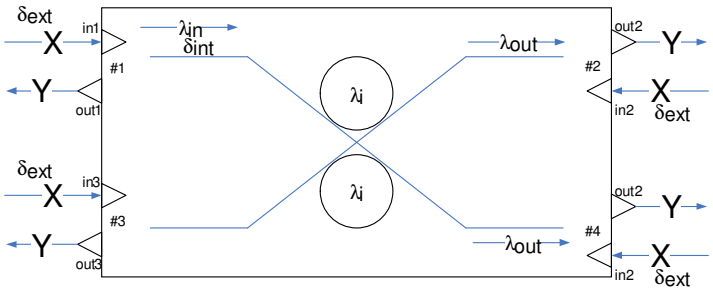


Figure 12. Optical switch with DEVS notations

$DEVS_{OS} = (X, Y, S, \delta_{int}, \lambda, ta)$

(15)

with:

inputs: $InPorts = \{'in1', 'in2', 'in3', 'in4'\}$

input set: $X = \{(p,v) | p \in InPorts, v \in X_p\}$

with: $X_p = wave\_value \in \{wavelength, power\}$

and,

output: $OutPorts = \{'out1', 'out2', 'out3', 'out4'\}$

output set: $Y = \{(p,v) | p \in OutPorts, v \in Y_p\}$

with: $Y_p = wave\_value \in \{wavelength, power\}$

The *states* are: $S = \{'idle', 'communication'\} \cdot InPorts$ (16)

The *internal events* are:

$\delta_{int}(phase, \sigma, inport, wave\_value, wavelength\_OS): S \rightarrow S$

$= ('busy', \sigma, p, v)$

if $phase = 'communication'$ and $p \in InPorts$ and $v \in X_p$

$= ('idle', \sigma, p, v)$ else.

The *external events* are:

$\delta_{ext}(phase, \sigma, e, x): Q \times X \rightarrow S$

$= ('idle', e, p, v)$

if $phase = 'idle'$ and $p = wave$ and $v = 0 \cdot exp(0)$

$= ('in\_light', process\_time, p, v)$

if $phase = 'communication'$ and $p = wave$ and $v = wave\_value \cdot OSdefects$

with $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$

The *output functions* are:

$\lambda(phase, \sigma, inport, wave\_value, wavelength\_OS): S \rightarrow Y$

$= (out2, wave\_value \cdot OSdefects)$

if *phase = 'communication'* and *wave_wavelength_value = wavelength_OS* and *inport = in1*

= (out4, wave_value·OSdefects)

if *phase = 'communication'* and *wave_wavelength_value ≠ wavelength_OS* and *inport = in1*

= (out1, wave_value·OSdefects)

if *phase = 'communication'* and *wave_wavelength_value = wavelength_OS* and *inport = in2*

= (out3, wave_value·OSdefects)

if *phase = 'communication'* and *wave_wavelength_value ≠ wavelength_OS* and *inport = in2*

= (out4, wave_value·OSdefects)

if *phase = 'communication'* and *wave_wavelength_value = wavelength_OS* and *inport = in3*

= (out2, wave_value·OSdefects)

if *phase = 'communication'* and *wave_wavelength_value ≠ wavelength_OS* and *inport = in3*

= (out3, wave_value·OSdefects)

if *phase = 'communication'* and *wave_wavelength_value = wavelength_OS* and *inport = in4*

= (out1, wave_value·OSdefects)

if *phase = 'communication'* and *wave_wavelength_value ≠ wavelength_OS* and *inport = in4*

= (out,'X') if *phase = 'idle'* with *out∈OutPorts*

The *state advancing time* is:

$ta(\sigma): S \rightarrow \Re^{+}_{0,\infty} = \sigma = bit\_propagation\_time$ (17)

Two states characterize the 4-port optical switch behavior, as shown in (16): *idle* (no conversion) and *communication* (light is routed through the optical switch, either in the

cross state or either *bar* state as seen in Figure 13). There are 2 internal events: *busy* (light is present), *idle* (no light through the switch); and 2 external events: *idle* (no light) and *in_light* (light in one of the inputs). The state advancing time is shown in (17) and is due to the light routing in the microresonator and in the waveguide depending on its geometry and its manufacture materials. As for the point to point connection, this DEVS description must take into account the attenuation in the switch due to its defects (OSdefects).

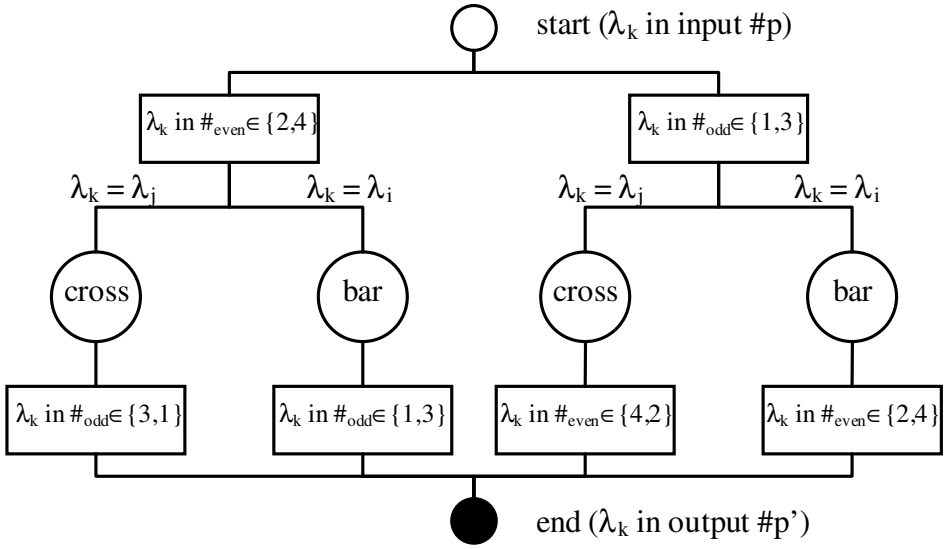


Figure 13. State diagram of a 4-port optical switch

Figure 13 presents the state flow of a 4-port optical switch. This diagram takes into account the DEVS events seen previously.

2.3.3 4 X 4 λ-Router

Figure 14 presents the DEVS description of a 4 × 4 λ-router ([53], [54], [55]). To simplify the read, and since the 4 × 4 λ-router behavior is a combination of point to point connection and 4-port optical switch behavior (as defined in sub-section 2.3) a state diagram is only shown in Figure 15. *Ci* represents the connection between any input ports with the i^{th} output port.

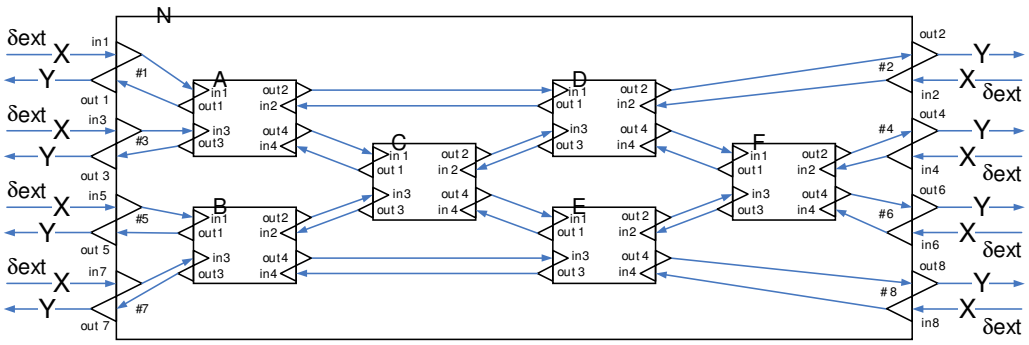


Figure 14. Optical switch with DEVS notations

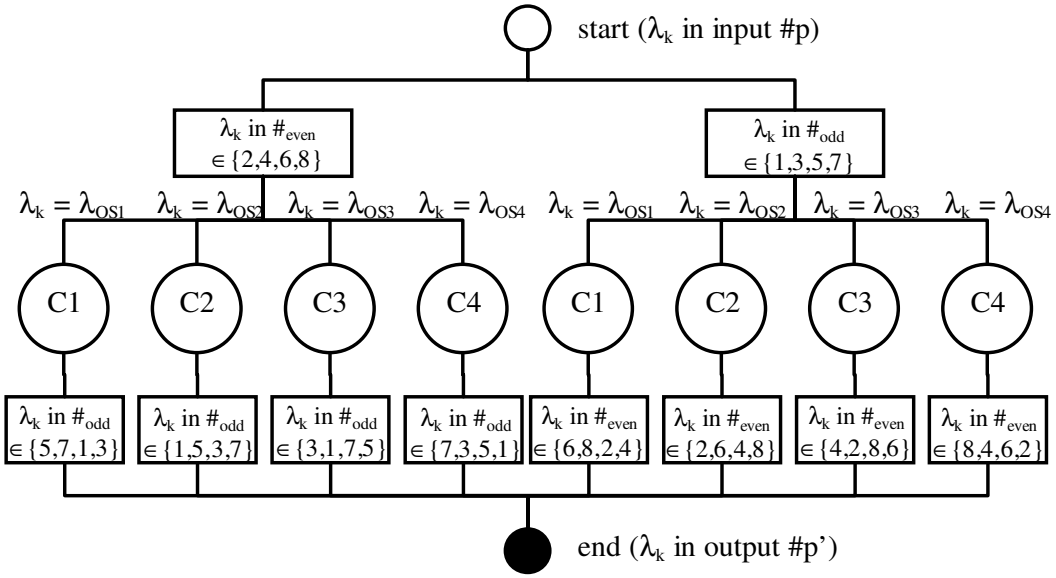


Figure 15. State diagram of a 4 x 4 λ-router

3 Modeling and Formal Verification for the Global Validation of the Behavior of a Passive ONoC

This section presents the modeling and the formal verification for the global validation of the behavior of a passive optical network on chip. The model was realized using timed-automata and was validated through simulation using UPPAAL toolbox. Its formal verification was realized by defining and checking its main properties. One of

the most important characteristics of the optical network is the non contention. This particularity requires complex models: the number of timed-automata increases and by consequence the verification becomes time consuming. To cope with this complexity, the modeling and the formal verification were realized in two steps. The first step consists of the modeling of the behavior of the network at high level of abstraction. For the second step, the abstraction level was lowered and the formal verification was realized on segments of the network. By doing so, the deadline verification time was reduced from more then 12 hours to 41 sec.

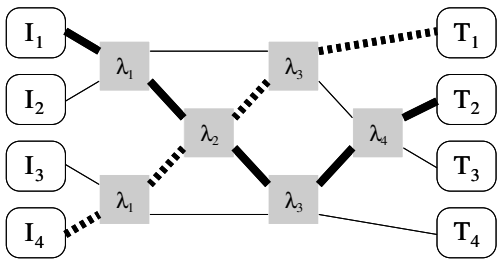
5.4.1 Optical Network on Chip Modeling

Contention occurs in a network when two nodes attempt to access a communication channel at the same time. The contention-free property of the optical network on chip increases the complexity of the modeling process. Thus, modeling the transmission of different wavelength in the same time requires a larger number of automata. This makes ONoC models very complex, comparing to other models representing an electrical network for instance that does not provide parallelism.

The routing in the optical network presented here is realized by a *4 X 4* λ-router (as presented in Figure 16 ([53], [54] and [55])). In order to model and validate its behavior we used the timed automata ([46] and [47]) and the UPPAAL tool [48].

I/T	T_1	T_2	T_3	T_4
I_1	λ_2	λ_3	λ_1	λ_4
I_2	λ_3	λ_4	λ_2	λ_1
I_3	λ_1	λ_2	λ_4	λ_3
I_4	λ_4	λ_1	λ_3	λ_2

(a) Truth table



(b) 4X4 λ-router

Figure 16. *4 x 4* λ-router

Due to the parallelism that is expected in an optical network, the system is represented with 44 processes (and consequently 44 automata), divided in subsystems as follows: four to represent the initiators, 16 for the targets (for each target in Figure 16(b) we needed four processes, one for each wavelength) and 24 for the routing structure. One of the most useful properties to check in a system is reachability meaning that one wants to check if all the states of an automaton are reachable, meaning for our model that we need to check that there exists an execution starting at the initial state that is the set of initiators {I1, I2, I3, I4} and reaching all the targets {T1, T2, T3, T4} for all the wavelength {λ1, λ2, λ3, λ4}. Our experiments showed that the verification of the reachability for this implementation becomes costly in terms of time and can take more than 12 hours because of the state explosion. Therefore, in order to improve the performances of the optical network model, its modeling and formal verification were realized in two steps:

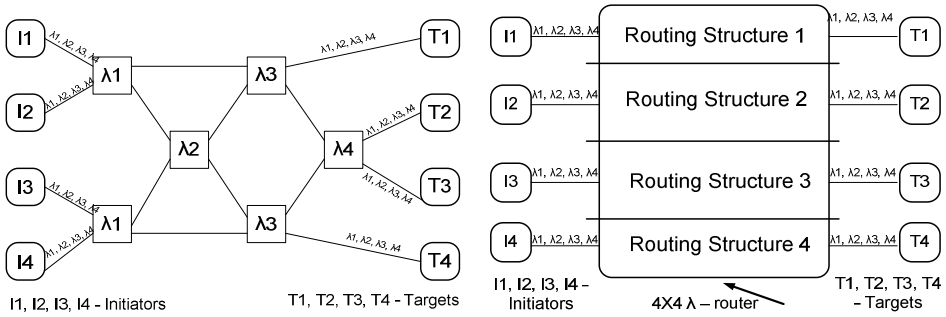
- The first step consists of modeling and verification of the global network and for this representation we raised the level of abstraction.
- The second step consists of modeling and verification of the behavior of the router at a lower level of abstraction when only one initiator and four targets are used.

This methodology allowed the verification of the contention in the global network, between initiators and mode detailed between the different signals generated by the same initiator when the signals have different wavelengths.

The complete checking takes only 2 seconds for the first step and 41 seconds per initiator for the second step. As one can see the verification time is drastically reduced using the proposed approach. Next sections detail these two steps.

Global model for 4 X 4 λ-router

Figure 17(a) shows the global network at its initial level of abstraction (as a set of four switches and Figure 17(b) shows the equivalent λ-router at a higher level of abstraction.



(a) *4 X 4* λ-router architecture (b) Equivalent abstract λ-router

Figure 17. Block schema of the passive optical 4 x 4 λ-router

The abstract router (shown in Figure 17(b)) is modeled as a set of four processes also named here Routing Structures. The four processes model the parallelism provided by the optical network: all the initiators can send data concurrently and all this data will be routed in parallel to the targets by the λ-router. Due to this parallelism, the same target can receive data from the four initiators in the same time. To respect this behavior the abstract router has four inputs (one from each initiator) and 16 outputs (four for each target). Each routing structure connects one initiator with the wavelength corresponding targets. Furthermore, the model has to verify the truth table shown in Figure 16(a), therefore, each target has to have four inputs, one for each wavelength.

As a result, the global model of the *4 X 4* λ-router is represented using 24 processes: four processes are used for the initiators, 16 for the targets (as they were previously explained) and four processes for the routing structure, one for each initiator.

Figure 18 shows the timed-automata model, in UPAAL, for one of the four parallel routing structures that connects an initiator to the targets. The left pane presents all 24 processes. The model has only one initial location (a double circle in Figure 18) *Start*. The router will change location from *Start* to *ReceiveDataFromInitiator(n)* (where is the number of the initiator from 1 to 4) following the transition $\text{Start} \xrightarrow[\text{lambda : int[lambda 1, lambda4]}]{\text{DataToSwit ch?}} > \text{ReceiveDat aFromIniti ator}$. This transition is realized with zero time and it is triggered by the receiving of the data (that is also synchronization between the

initiator and the router) from the initiator (`DataToSwitch?`). The transition also allows the random selection of a wavelength between the four wavelengths of the network λ1, λ2, λ3 and λ4, using `lambda:int[lambda1,lambda4]`. The location changes then to one of locations *ToTarget1, ToTarget2, ToTarget3, ToTarget4,* depending of the lambda selection. Each of this transition to a different target is determine by the value of lambda and for each transition there is synchronization DataToTarget! between the router and the corresponding target. The data is received by the corresponding target and the simulation context changes to the processes named here `Target` that are identified by different indexes. Each one of these processes receives data from the router (`DataToTarget?`).

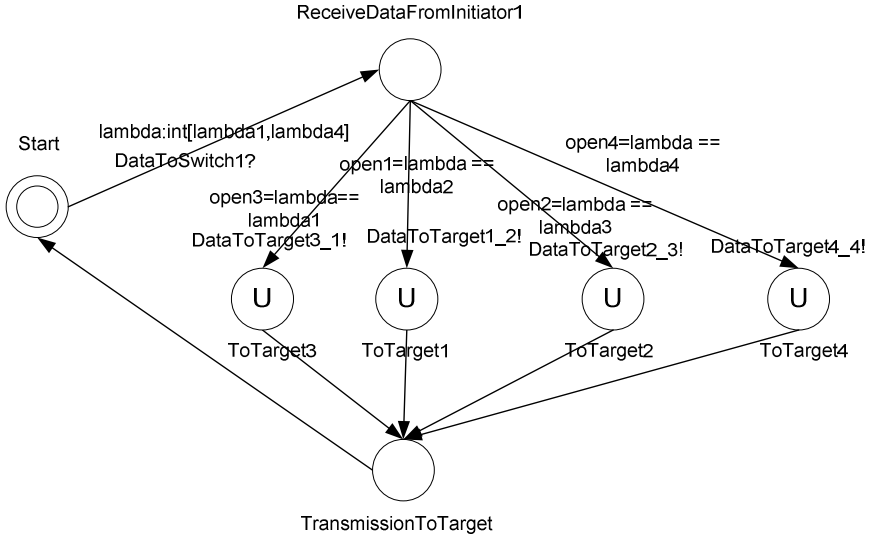


Figure 18. Routing structure representation

Figure 19 presents a screenshot with the simulation of the abstract λ-router. This figure shows the parallelism between the different targets (Target1, Target2 and Target3) and the parallelism between signals of different wavelength (λ3, and λ4) in the same target – Target1.

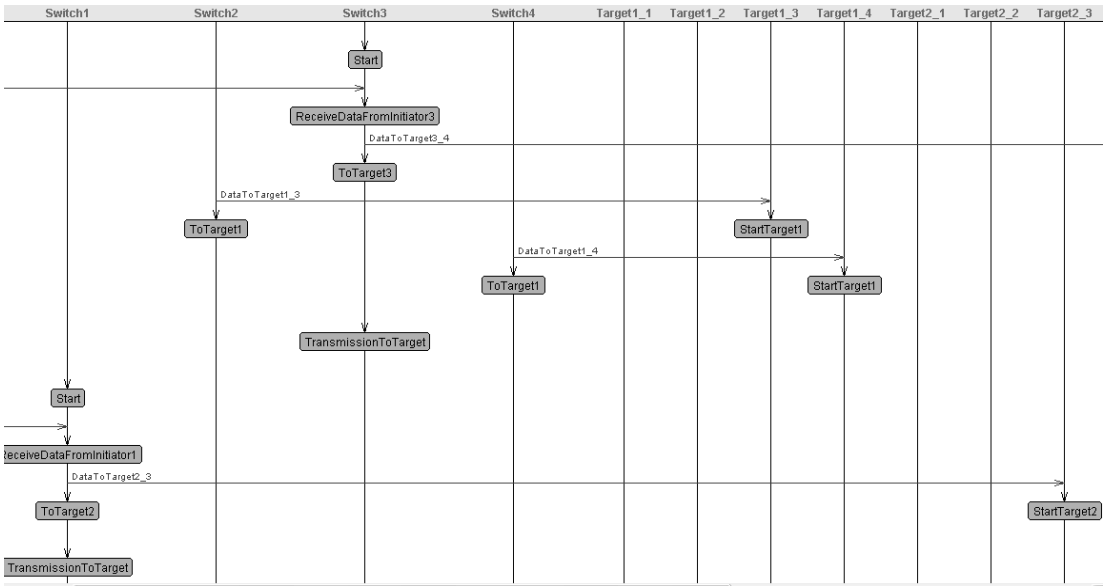


Figure 19. λ-router simulation screen capture

Model for a signal path generated by one initiator

Figure 20 shows the model of the path of the signal generated by one initiator (in this example the initiator I1) at its original level of abstraction. The signal is routed through the four λ-routers in order to reach the designated targets. The dashed lines and λ-routers represent the paths that are not reached by the signal sent by the initiator I1. Moreover, the model verifies the truth table presented in Figure 16(a) and Table 2. In the first step I1 can send to the λ-router λ1 four signals corresponding to four different wavelengths. Here the signal corresponding to the wavelength λ1 is sent to λ3 and the remaining three signals are sent to the λ-router λ2 where a new selection is made.

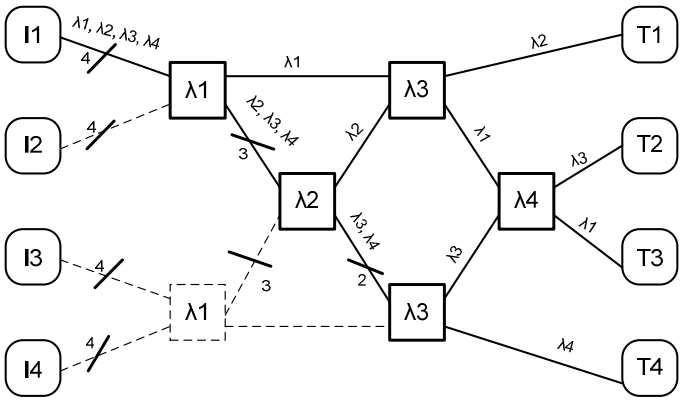


Figure 20. Signal path in the 4 x 4 λ-router for one initiator

As shown in Figure 20, in order to represent the exact path of the signal from the initiator to the targets, the model requires 12 processes: one for the initiator, 4 for the targets (each one with its own wavelength) and seven processes for the routing. The UPPAAL representation for this model is similar with the one where all initiators were represented. The simulation of the second step of the passive ONoC showed the parallelism between the different signals of different wavelength in the same switch.

5.4.2 Optical Network on Chip Formal Verification

Using UPPAAL, the models were simulated and formally verified. This is a verification of the functionality of the models.

Formal verification of global 4 X 4 λ - router

The following properties were verified for the global model where all the routers were abstract into one router that summed the behavior of the whole network.

P0 Absence of deadlock (safety property)

Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.

```
A[] not deadlock
```

P1 Absence of contention in the global model (reachability property)

Definition: simultaneous wavelength can be sent through the network in the same time.

```
E<>            Switch1.TransmissionToTarget            and
Switch3.TransmissionToTarget                           and
Switch3.TransmissionToTarget                           and
Switch4.TransmissionToTarget
```

P2 All locations in the automaton representing the switch are eventually taken (liveness property)

Definition: whenever a wavelength takes the *ReceiveDataFromInitiator1* location in the Switch1, it will eventually take the *TransmissionToTarget* location in the same switch.

```
A<>         Switch1.ReceiveDataFromInitiator1          imply
Switch1.TransmissionToTarget
```

P2 Verification of the truth table (safety property)

Definition: there is one and only one wavelength that connects one initiator with one target (truth table in Figure 16(a)). We give here the syntax for only one of the initiators; the properties for the other three were verified in the same manner.

```
A[] Switch1.TransmissionToTarget and lambda==lambda1 imply
Target3_1.StartTarget3
A[] Switch1.TransmissionToTarget and lambda==lambda2 imply
Target1_2.StartTarget1
A[] Switch1.TransmissionToTarget and lambda==lambda3 imply
Target2_3.StartTarget2
A[] Switch1.TransmissionToTarget and lambda==lambda3 imply
Target4_4.StartTarget4
```

Formal verification of a 4 X 4 λ - router

The following properties for the model where the signal generated by one initiator is routed through all λ– routers that form the optical network were verified:

P0 Absence of deadlock (safety property)

```
A[] not deadlock
```

P1 Absence of contention in the network (reachability property)

Definition: simultaneous wavelength can be sent through the router, from the same initiator, in the same time. For one initiator the parallelism in the same switch is encountered in the switches with λ3 and λ4. We verified here the parallelism for both situations:

```
E<>                Switch3a_1.DataToTarget1                and
Switch3a_2.DataToSwitch4 and Switch3b.DataOutSwitch3b
E<> Switch4_1.DataToTarget3 and Switch4_2.DataToTarget2
```

P2 Verification of the truth table (safety property)

Definition: the truth table shown Figure 16(a) was also verified for one initiator. This property validates also the connection between one initiator and four targets.

```
A[] Switch1.TransmissionToTarget and lambda==lambda1 imply
Target3.StartTarget3
A[] Switch1.TransmissionToTarget and lambda==lambda2 imply
Target1.StartTarget1
A[] Switch1.TransmissionToTarget and lambda==lambda3 imply
Target2.StartTarget2
A[] Switch1.TransmissionToTarget and lambda==lambda4 imply
Target4.StartTarget4
```

5.5 Conclusion

In this chapter we proposed a novel approach that enables: the possibility to formalize very recent technologies using DEVS approach and the use of this formalization to validate and debug complex systems as optical-electrical interfaces and the modeling, the simulation and the formal verification for the global validation of the behavior of a passive integrated photonic routing structure using models that are based on timed

automata. We presented the formalization of three types of blocks that form a ONoC: a transmitter interface circuits (for the electro-optical conversion), a passive integrated photonic routing structure (named λ- router) and a receiver interface circuit (for the opto-electrical conversion). The formalization was then completed with the modeling, the simulation and the formal verification of a passive integrated photonic routing structure. The modeling as well as the simulation and the formal verification were divided in two steps. The first step consisted of the verification of the global *4 X 4* λ – router at a high level of abstraction, as one router behavior while the second step was the representation at a lower level of abstraction of one initiator and the signal path through the optical network. Formal properties were defined and checked for both models. The complete checking takes only 2 seconds for the first step and 41 seconds per initiator for the second step. As one can see the verification time is drastically reduced using the proposed approach.

ANNEX 2 – CODIS FRAMEWORK

CODIS is a tool which can automatically produce the global simulation model instances for discrete/continuous systems simulation using SystemC and Simulink® simulators. This is done by generating and providing interfaces which implement the simulation model layers and building the co-simulation bus. Figure 21 gives an overview of the flow of the instances generation in the case of CODIS ([51], [52]).

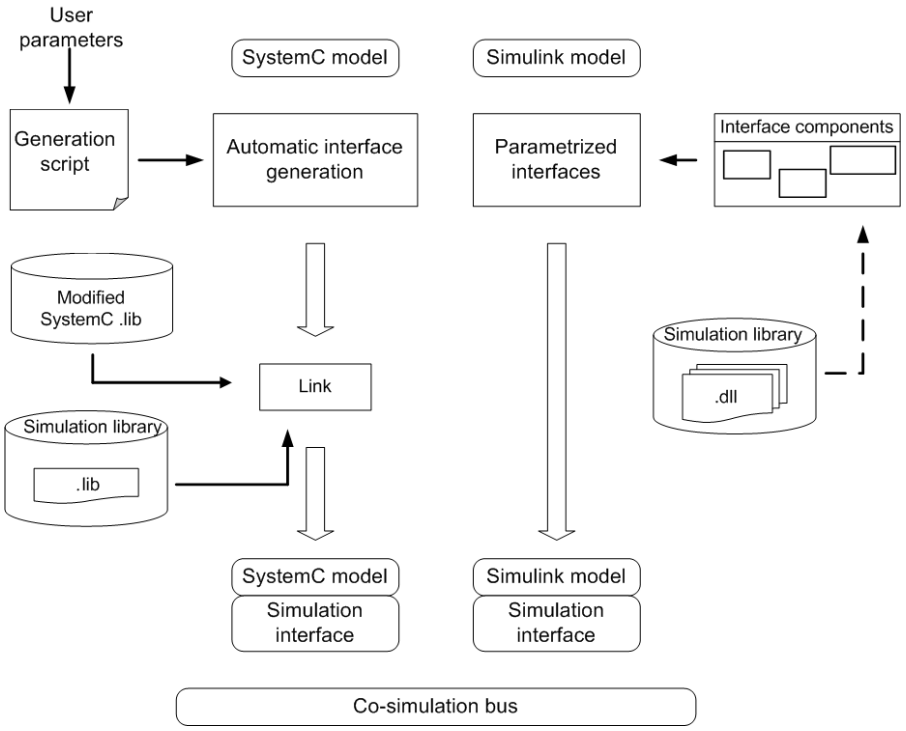


Figure 21. Overview of the CODIS flow

The inputs in the flow are the continuous model in Simulink® and the discrete model in SystemC which are, respectively, schematic and textual models. The output of the flow is the global simulation model (co-simulation model) instance. For Simulink®, the interfaces can be parameterized starting with their dialog box. The parameters of Sim_inter_In and Sim_inter_Out interfaces are the number of input and respectively output ports. State interface has a parameter defining the state events number. The user starts by dragging the interfaces from the interface components library into his model's

window, then parameterizes them and finally connects them to the inputs and the outputs of his model. Before the simulation, the functionalities of these blocks are loaded by Simulink® from the .dll libraries. For SystemC, the SC_inter_In parameters are: (1) the names, the number and the data type of the discrete model inputs ports and (2) the sampling periods. The SC_inter_out parameters are the names, the number and the data type of the discrete model outputs ports. The interfaces are automatically generated by a script generator that has as input the user-defined parameters. The tool generates also the function sc_main (or modifies the existing sc_main) that connects the interfaces to the user model. The model is compiled and the link editor calls the library from SystemC and a static library (the simulation library in Figure 21). The implementation was performed in the case of SystemC as a discrete event simulator and Simulink® as a continuous simulator.

In this annex we detail the model of interaction between the continuous and the discrete simulators and present the interfaces implementation. One must note that the interfaces between SystemC and Simulink® have been previously proposed for pure digital systems [58] but not for discrete–continuous systems. For a better explanation, we start by presenting briefly the SystemC and Simulink® simulators.

1. SystemC Simulator

SystemC [8] simulator is an effective and relatively simple scheduler. Its task is to determine processes execution order by considering their sensitivity lists and events time stamps. Events are ordered in a special queue and classified into two types: zero-delayed and timed events. The scheduler uses the notion of delta cycle. At a particular discrete time, multiple delta cycles may occur until the simulated model becomes stable: no signals to change, or in a general way, no more zero-delayed events to consider at the current time. Then, the scheduler consults its queue to extract the next event (next discrete time) if any, otherwise it stops. This cycle is repeated until the end of simulation.

2. Simulink Simulator

Simulink® [16] simulator solves system equations and updates states and outputs of blocks once per integration step, which can be fixed or variable. The order in which the blocks are updated is critical for results validity. If the block's outputs are a function of its inputs, the block must be updated after the blocks that drive its inputs (e.g. adder or gain computing block). Simulink uses minor and major steps. Minor step are used to improve the accuracy of result at major steps. Signals are updated only at major steps.

3. The Simulation Interfaces

Figure 22 shows the continuous and the discrete models with the simulation interfaces. The interfaces implement the co-simulation layers. They represent the software components required to integrate the two simulators with respect to the simulation model. For Simulink®, the interfaces are S-functions blocks. These blocks are manipulated like all other components of the Simulink® library. They contain input/output ports compatible with all model ports, which can be connected directly by using Simulink® signals.

They are classified into four types:

- The Sync interface implements the critical part of the ‘‘Discrete events detection’’ layer. It creates break points, which must be reached accurately by a solver (a variable step solver). These points are the time stamps of the received events (sampling events or signals update events). When an event is received, this interface makes its next activation time equal to this event time stamp. Once this time stamp is reached, the Sync is executed to set its next activation time equal to the new received event time stamp, etc. The interface is executed at t = 0 to fix its first activation time.

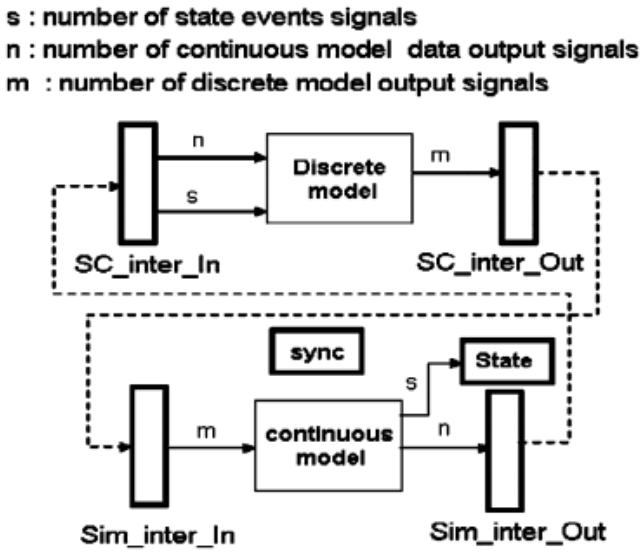


Figure 22. Continuous and discrete models integrating the co-simulation interfaces

- The Sim_inter_In interface implements the communication layer (input function), the ‘‘Context switch’’ layer and a part of the ‘‘Discrete events detection’’ layer, which is responsible in detecting the passage of the solver by the time stamps of the sampling events (breakpoints). Once this passage is detected, the interface switches the simulation context and executes the communication layer (reads signals).

- The Sim_inter_Out interface implements the communication layer (output function), the ‘‘Context switch’’ layer and a part of the ‘‘Discrete events detection’’ layer, which is responsible in detecting the passage of the solver by the sampling events time stamps (breakpoints). Once this passage is detected, the interface executes the communication layer (sends signals) and switches the simulation context.

- The State interface implements the ‘‘Detection and sending of sate events’’ and the ‘‘Context switch’’ layers.

For SystemC, the interfaces are programmed as SystemC modules. They are classified into two types:

- The SC_inter_In interface implements the input communication function and ensures synchronization with input data and state events. It can be viewed as a sampler circuit

and can be auto clocked or have an external clock supplied by the discrete model. The interface has two types of signals:

- Data signals, which are sc_signal or sc_fifo type. If the discrete model input ports are bits vectors then the interface add functionality converting double data to bit vector data.

- State events signals, which are boolean type (bit). Each time the continuous simulator sends a state event, the corresponding state event signal is set to ‘‘1’’.

- The SC_inter_Out interface implements the output communication function and ensures synchronization with output data. If the discrete model output ports are bits vectors then the interface add functionality that converts bits vector data to double data.

4. The Interaction Between SystemC and Simulink®

Simulink® interacts with SystemC through its interfaces. These interfaces and the user model’s blocks are executed at each integration step. The execution order respects the data dependency rule. SystemC interacts with Simulink® through its interfaces and its scheduler. The scheduler integrates the ‘‘End of discrete simulation cycle detection and events sending’’ layer and the ‘‘State events consideration’’ layer.

5. Interfaces Implementation

Example of Simulink® interfaces

For Simulink®, the interfaces are S-functions programmed in C++. An S-function is programmed using a number of predefined functions. In our case, five functions are used. The user adds its code to these predefined functions. For example, a code used to initialize the simulation is added to the MdlInitializeSizes function, a code used to compute output signals is added to the MdlOutputs function, etc. The pseudo-code of two interfaces is given by Figure 23 and Figure 24 (only the principal functions are shown).

```
static void mdlInitializeSampleTimes(...)
{
  /* set the time mode of the S-function */
 ssSetSampleTime(S,0,VARIABLE_SAMPLE_TIME);
    ...
}

static void mdlGetTimeOfNextVarHit(...)
{
Now = ssGetT(S);
...
Next_break_point = Max (last_sampling_event
_timestamp, last_signals_update_events
_timestamp);
/* set the next execution time of the S-
function */
ssSetTNext(S, Next_break_point);
}

static void mdlOutputs(...)
{
   /*nothing to do */
}
```

Figure 23. Sync interface pseudo-code

```
static void mdlInitializeSampleTimes(...)
{
 ssSetSampleTime(S,0,  INHERITED_SAMPLE_
TIME);
 ssSetOffsetTime(S, 0, 0.0);
}

static void mdlOutputs(...)
{
 double Now = ssGetT(S);
 if( (Now == sampling_event_timestamp &&
ssIsMajorTimeStep(S))
 {
   for(int i=0; i < N_Out_Port; i++)
   {
   /* update output signals to SystemC */
   *((double*)SharedMemoryPtr + 50 + i) =
*(*ssGetInputPortRealSignalPtrs(S,i));

/* Context Switch */
  if(Now != 0)
  {
   /* release SystemC */

  if (!ReleaseSemaphore(...)
  {
  printf("Can not release signal sem);

  }
  /* stop Simulink */
  WaitForSingleObject(
  hSemClient,    // handle to semaphore
  INFINITE);     // no time-out
}}}
```

Figure 24. Sim-Inter_Out interface pseudo-code

In Figure 23 `sync` interface uses a special time mode, which is the variable_sample_time. With this mode, one can choose the next execution time of the S-function equal to the next discrete event sent by SystemC synchronization layer. In this case, Simulink® adjusts the integration steps to satisfy the criteria of resolution and to reach with exactitude the time execution of this S-function (which is the time stamp of the SystemC event).

SystemC interfaces

The interfaces are implemented as SystemC modules programmed in C++. For each interface, this sub-section gives the .h and .cpp files, classically used to describe SystemC modules.

```
#define et_mat0 sc_get_curr_simcontext()->
et_mat[0] /* this definition is in the file
defining the environment variables added
for heterogeneous simulation */

SC_inter_In.h
SC_MODULE(interIn)
{
/* data input ports */
sc_out <double> data0, data1;
/* State events input ports */
sc_out <sc_bit> StateEventPort;
/* sampling clock */
sc_in <bool> clk;

void In_data (); /* communication layer */
void state (); /* part of the layer "State
events consideration" */

SC_CTOR(SC_inter_In)
{
      SC_METHOD (In_data);
      Sensitive_pos(clk);

/* creation of et_mat0 event associated
with state event */
      et_mat0 = new sc_event;
/* make the process 'state' sensitive to
et_mat0, as consequence to the state event
*/
      SC_METHOD(state);
      sensitive(et_mat0);
}};

SC_inter_In.cpp
Void SC_inter_In :: In_data ()
{
/* update input signals */
data0.write(ReadSignalFromSimulink(0));
data1.write(ReadSignalFromSimulink(1));
}
Void SC_inter_In :: state()
{
 StateEvPort.write(~StateEventPort.read());
}
```

Figure 25. SC_inter_In interface code

Examples of SC_inter_In and SC_inter_Out interfaces are given in Figure 25 and Figure 26 respectively.

SC_inter_Out.h

```
SC_MODULE(interOut)
{
   /* output ports */
   sc_in <double> data, noise;

   void send_data();/*communication layer*/
       SC_CTOR(interOut)
       {
              SC_METHOD(send_data);
              sensitive << data << noise;
              dont_initialize();
       }
};
```

SC_inter_Out.cpp

```
void interOut :: send_data()
{
 /* update output signals */
 WriteSignalToSimulink(data.read(), 0);
 WriteSignalToSimulink(bruit.read(),1);
}
```

Figure 26. SC_inter_Out interface code

PUBLICATIONS

1. L. Gheorghe, G. Nicolescu, H. Boucheneb: "Semantics for Rollback-Based Continuous/Discrete Simulation" IEEE Int'l Behavioral Modeling and Simulation Conference, BMAS 2008.
2. E. Bensoudane, D. Tonietto, L Gheorghe, G. Nicolescu: "System-Level Design of Continuous/Discrete-Time Heterogeneous Systems Applied to High-Speed Serial Link" IEEE Newcas-Taisa Conference, 2008
3. L. Gheorghe, F. Bouchhima, G. Nicolescu, H. Boucheneb: "Semantics for Model-Based Validation of Continuous/Discrete Systems" in Design Automation and Test in Europe (DATE'08), 2008.
4. M. Briere, L. Gheorghe, G. Nicolescu, I. O'Connor, G. Wainer: "Towards the High-Level Design of Optical Networks on-Chip. Formalization of Opto-Electrical Interfaces" in IEEE Int'l conference on Electronics, Circuits and Systems, 2007.
5. L. Gheorghe, F. Bouchhima, G. Nicolescu, H. Boucheneb: "A Formalization of Global Simulation Models for Continuous/Discrete Systems" in Proc of the 2007 Summer Computer Simulation Conference (SCSC'07), 2007.
6. G. Nicolescu, H. Boucheneb, L. Gheorghe, F. Bouchhima: "Methodology for efficient design of continuous/discrete-events co-simulation tools" High Level Simulation Languages and Applications (HLSLA'07), 2007.
7. L. Gheorghe, F. Bouchhima, G. Nicolescu, H. Boucheneb: "Formal definitions of simulation interfaces in a continuous/discrete co-simulation tool" In: Proc. IEEE Workshop on Rapid System Prototyping (RSP'06), 2006.
8. G. Nicolescu, F. Bouchhima, L. Gheorghe: "CODIS-A framework for continuous/discrete systems co-simulation" 2nd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'06), 2006

9. L. Gheorghe and G. Nicolescu: “MP SoCs incuding optical interconnect. Technological progresses and challenges for CAD tools design” at International Workshop SoC for Real Time Applications (IWSOC’05), 2005

Book Chapters

10. F. Bouchhima, L. Gheorghe, G. Nicolescu, E. Aboulhamid, M. Abid: “The anatomy of a continuous/discrete execution model for timed execution heterogeneous systems” in “Global Specification and Validation of Embedded Systems” Springer, 2007 .
11. L. Gheorghe, G. Nicolescu, H. Boucheneb: ”A Generic Methodology for the Design of Continuous/Discrete Co-simulation Tools” in “Model-based design for Embedded Systems” CRC Press, to appear in 2009
12. L.Gheorghe, G. Nicolescu, I. O’Connor: “Modeling and Validation of Optical Networks on Chip” in “Discrete Event System Specifications - DEVS” CRC Press, to appear in 2009

Submitted Papers

13. L. Gheorghe, F. Bouchhima, G. Nicolescu, H. Boucheneb: “A Generic Methodology for the Design of Continuous/Discrete Simulation Tools” submitted to IEEE Transactions on Computers.
14. L. Gheorghe, G. Nicolescu, I. O’Connor: “Modeling and Formal Verification of a Passive Optical Network on Chip Behavior” submitted to Design Automation Conference DAC’09.
15. B. Girodias, L. Gheorghe, Y. Bouchebaba, G. Nicolescu, E. Aboulhamid, M. Langevin, P. Paulin: “Combining Memory Optimization with Mapping of Multimedia Applications for Multiprocessors System on chip” submitted to ISSS CODES 2009

16. B. Girodias, L. Gheorghe, Y. Bouchebaba, G. Nicolescu, E. Aboulhamid, P. Paulin , M. Langevin, "Integrating Memory Optimization with Mapping Algorithms for Multi-Processors System-on-Chip" submitted to TECS Journal.