Modélisation, simulation et vérification de circuits numériques asynchrones dans le standard SystemC v2.0.1

A. Sirianni

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

|__/__/__/__/__/__/__/__/__/__|

THESE

pour obtenir le grade de

DOCTEUR de l'INPG

spécialité : **Micro et Nano Electronique**

préparée au laboratoire **TIMA** dans le cadre de

l'**Ecole Doctorale Electronique, Electrotechnique, Automatique et Traitement du Signal**

présentée et soutenue publiquement
par

Antoine SIRIANNI

le 18 juin 2004

Titre :

Modélisation, simulation et vérification de circuits numériques asynchrones dans le standard SystemC v2.0.1

Directeur de thèse : Marc RENAUDIN

JURY

M. Ahmed JERRAYA, Président

M. Eric MARTIN, Rapporteur

M. Bruno ROUZEYRE, Rapporteur

M. Marc RENAUDIN, Directeur de thèse

M. Laurent FESQUET, Co-directeur de thèse

M. Habib MEHREZ, Examinateur

Remerciements

Tout d'abord, je tiens à saluer ma famille et mes amis. Sans leur soutien, sans doute ne seriez-vous pas dans la situation de lire ce manuscrit, ni moi-même d'écrire ces mots, tant la vie réserve parfois des épreuves difficiles. Ceux dont je parle se reconnaîtront, mais je souhaite les désigner par leur nom. J'ai peur d'en oublier. Mais, voilà ceux à qui je pense aujourd'hui en particulier : Rosa et Savério, Hélène, Jean-François, Stéphane, Francesca, Francine, Lilianne et Joseph, Sylvianne et Ennio, Anna et Giuseppe, Marie et Gérard, Marie, Martine, Isabelle, Gabriela et Bogdan, Nadège et David, Pauline et Pierre, Nacer, Françoise et Ivano.

Ensuite, je remercie Marc Renaudin. Marc Renaudin est Professeur des Universités à l'Institut National Polytechnique de Grenoble (INPG). Il dirige avec énergie et brio le groupe CIS (« Concurrent Integrated Systems » en anglais) au sein du laboratoire des Techniques de l'Informatique et de la Microélectronique pour l'Architecture des Ordinateurs (TIMA). Marc a accepté avec élégance de diriger et de financer mes recherches. La richesse de l'expérience que j'ai acquise pendant ces années passées au sein de ce groupe de recherche est sans égale, sinon ma gratitude.

Je remercie particulièrement Laurent Fesquet, Maître de Conférences à l'INPG. Il a accepté d'assister Marc Renaudin dans la direction de mes recherches. Il a joué un rôle décisif dans la maturation du manuscrit grâce à sa disponibilité, ses idées et son ouverture d'esprit.

Puis, je remercie Pierre Gentil. Pierre Gentil est Professeur des Universités à l'INPG. Il dirige l'Ecole Doctorale Electronique, Electrotechnique, Automatique et Traitement du Signal (EEATS) de l'INPG. Il est également responsable des Etudes Doctorales dans la spécialité Microélectronique. Il m'a fait la faveur de m'inscrire en première année de thèse alors que ma formation ne le permettait pas de prime abord.

Comment ne pas remercier Bernard Courtois, Directeur de Recherche au Centre National de la Recherche Scientifique (CNRS), Directeur du laboratoire TIMA ? Sans son accord, il m'aurait été impossible d'intégrer le groupe CIS et d'évoluer ainsi dans l'environnement de recherche de stature internationale que représente le laboratoire TIMA.

Il est impossible de conclure ces remerciements sans me tourner vers les nombreuses personnes que j'ai croisées avec plaisir pendant ces années ? Je ne peux pas les nommer toutes. Cependant, je tiens tout particulièrement à citer celles et ceux avec qui j'ai été amené à collaborer. Il s'agit de Kamel Slimani, Yann Rémond, Gilles Sicard, Menouer Boubekeur, Jean-Baptiste Rigaud, Anh Vu Dinh-Duc, Amine Rezzag et Joaõ Fragoso, Sébastien Fillon et Alexandre Chagoya.

Je remercie encore Alain Guyot. Alain Guyot est Maître de Conférences à l'INPG. Il a accepté de partager son bureau avec moi pendant ces années. Cela nous a conduit à sympathiser et à échanger de nombreux propos sur autant de sujets différents. Sa culture et sa gentillesse resteront dans ma mémoire pour longtemps.

Pour finir, je remercie Madame Dominique Borrione, Professeur de l'Université Joseph Fourier (UJF). Elle dirige le groupe VDS («Verification and Modeling of Digital Systems » en anglais) du laboratoire TIMA. Ses très nombreuses qualités continuent de forcer mon admiration.

La justesse s'incrit dans le temps,
Et la vérité dans l'instant.

Avant-propos

Philosophons un peu.

Dans le sens commun, le déterminisme est un principe fortement lié au principe de causalité. Un système est par définition un tout constitué de parties. Un système causal est par définition un système dont l'évolution à tout instant est complètement et exactement déterminée par un certain nombre de conditions antérieures. Si l'on se penche attentivement sur les définitions précédentes, il apparaît que ces concepts reposent implicitement sur la notion de temps, sur la notion de discrétisation des événements et sur l'ordonnancement de ces événements discrets dans le temps. Toutes ces notions sont fortement conditionnées par notre perception des phénomènes, notre cerveau traitant de manière consciente les tâches les unes après les autres. C'est ce qui leur confère un caractère de début, de fin et les fait apparaître en séquence. Toute conception, processus conscient de traitement de l'information est alors calqué sur ce modèle. Et pourtant, Edsger W. Dijkstra, un des contributeurs majeurs du siècle dernier dans le monde de l'informatique écrivait que finalement, il préférait considérer le déterminisme comme un cas particulier du non déterminisme. C'est vrai que si l'on considère le comportement d'un système de manière statistique, il est possible de concevoir que son évolution à un instant donné est caractérisée par une variable aléatoire répondant à une densité de probabilité. En faisant tendre cette fonction vers une distribution, sous réserve de licité, on retrouve un comportement parfaitement déterminé et causal. Aura-t-on pour autant saisi le sens profond du non déterminisme? Il me semble plutôt que le non déterminisme est intimement lié au parallélisme. Et c'est sans doute ce qui rend cette notion difficile à saisir. Chacune des parties d'un système évolue de manière autonome, sauf lorsque l'une d'elles échange des informations avec le reste du système, alors perçu comme son environnement. Il apparaît alors clairement qu'à un instant donné plusieurs actions sont possibles en différents lieux du système, en dehors de toute considération statistique. Ces actions sont soit internes à une partie, soit nécessitent la collaboration dans le système d'une partie ainsi que de son environnement. Dans ce dernier cas, on parle de rendez-vous puisqu'à un instant donné, cette partie et son environnement se rencontrent pour échanger de l'information. C'est poétique, n'est-ce pas ? On dit aussi que la partie et son environnement se synchronisent puisqu'ils se retrouvent simultanément en un même lieu. C'est cette simultanéité qui caractérise la notion de synchronisme et rend possible l'échange d'information et donc la communication entre deux entités d'un système. Lorsque l'on cherche à réaliser physiquement un système, on peut choisir de construire une machine dont la structure est calquée sur la structure dudit système et on parle alors de parallélisme vrai. Cependant dans la plupart des cas, pour des questions de coût, les différentes parties se partagent les différentes ressources à disposition : mémoire, calcul, communication, temps. On parle alors d'entrelacement des actions dans le temps. Finalement, on peut voir toute conception comme un processus conscient de transformation visant à rendre séquentiel un système en éliminant progressivement toute possibilité sensible de choix.

Mais a-t-on le choix ?

Table des matières

Table des figures

Introduction

Après un rappel sur l'histoire des ordinateurs, nous définissons la problématique de la modélisation, de la simulation et de la vérification des circuits numériques asynchrones dans le standard SystemC v2.0.1 de conception de système numériques. Nous mettons en évidence notre contribution et nous présentons notre étude.

La crise des fondements

A la fin du XIX^{ème} siècle, les mathématiciens, dans un élan unificateur, cherchaient à établir les fondements des mathématiques. La théorie des ensembles n'était pas satisfaisante puisqu'elle conduisait à des paradoxes tel que celui de Bertrand Russell dont l'énoncé suit. *Si l'ensemble des ensembles ne se contenant pas eux-mêmes se contient lui-même, alors il ne se contient pas lui-même*. David Hilbert [1] pensait que ce défaut provenait d'un manque de précision dans la formalisation. Il devait exister un ensemble d'axiomes et de règles d'inférence pour engendrer toutes les mathématiques. Kurt Gödel prouva définitivement le contraire en 1931 à travers le théorème d'incomplétude de l'arithmétique [2]. Cela permit de résoudre le second problème de Hilbert. Il existe effectivement en arithmétique des conjectures, i.e. des formules dont l'intuition indique qu'elles sont vraies mais que l'on ne sait pas démontrer. Mais peut-on déterminer à l'avance l'existence d'une telle démonstration ?

La machine de Turing

Etudiant la calculabilité des nombres, Alan Turing introduisait en 1936 une machine abstraite constituée d'un ruban découpé en cases contiguës, d'une tête de lecture/écriture pouvant se déplacer vers la droite ou vers la gauche et d'une machine à états finis contrôlant les déplacements et les opérations de la tête de lecture [3]. Il formalisait ainsi pour la première fois le concept d'algorithme et prouva que le problème de savoir si une telle machine s'arrête un jour est indécidable. On ne peut donc prédire si une conjecture est démontrable avant d'en avoir fourni la preuve.

La plupart des nombres, i.e. les nombres entiers, les nombres rationnels et certains nombres réels « réguliers » comme les nombres transcendants sont calculables par cette machine. C'est ce que l'on a coutume de résumer sous la forme de puissance d'expression de la machine de Turing.

Ainsi la logique confirmait la nécessité de développer autant de théories pour traduire la réalité du monde. Les théories des ensembles, des modèles, de la démonstration, de la calculabilité, de la relativité, de la mécanique quantique et du signal établissaient progressivement le socle qui constitue aujourd'hui les technologies de l'information.

L'architecture de Von Neumann

A mesure que les barrières technologiques tombaient, de la lampe au transistor bipolaire, il a été possible de construire des machines de plus en plus complexes jusqu'à arriver à l'architecture de von Neumann [4] (voir Figure 1) en 1945, le langage assembleur étant inventé en 1950.

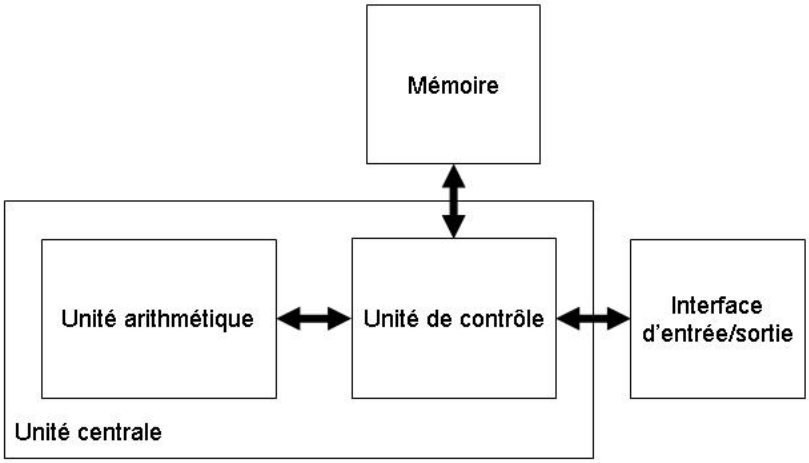


Figure 1 : Architecture de Von Neumann

L'architecture d'un système sur une puce

Depuis, avec l'avènement des technologies d'intégration CMOS, d'un côté, le niveau d'intégration n'a eu de cesse d'augmenter à un rythme exponentiel suivant la fameuse loi de Moore. De l'autre, sur le plan des langages de programmation, le niveau d'abstraction n'a cessé d'augmenter. On est aujourd'hui capable d'intégrer semi automatiquement un système complet contenant plusieurs dizaines de millions de transistors sur une puce. Le tout héberge des applications logicielles, un système d'exploitation, des fonctions électroniques analogiques et mixtes, des capteurs et des modules radiofréquences. Une architecture générique (voir Figure 2) pour le cœur d'un tel système est proposée dans [5]. Elle s'articule autour d'un bus de communication interconnectant un processeur, une mémoire, une interface d'entrée/sortie, un contrôleur d'accès direct à la mémoire, des minuteurs, des gestionnaires d'affichage. L'architecture de Von Neumann n'est pas si lointaine.

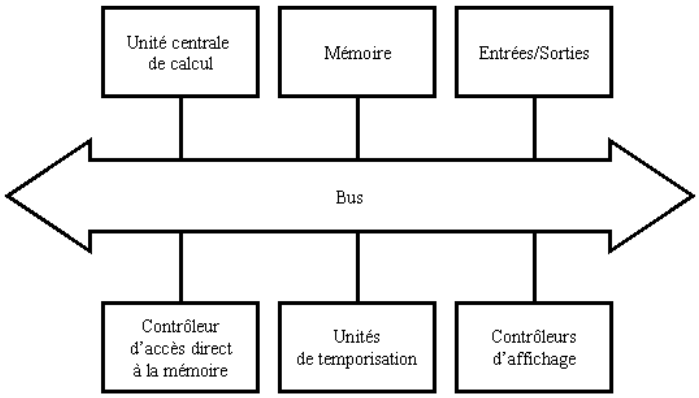


Figure 2 : Architecture générique d'un système sur une puce

Les recommandations de l'ITRS

Aujourd'hui, étant donné la variété et la complexité des problèmes à résoudre lors de la conception d'un système sur une puce, de nombreux acteurs de l'industrie du semi-conducteur coopèrent globalement à l'élaboration d'une « feuille de route », nommément l'ITRS (« International Technology Roadmap for Semiconductors » en anglais). Elle dresse les perspectives du domaine sur une période de quinze ans à partir de documents et de données actualisées suivant les avancées majeures et les nouvelles tendances dans l'industrie.

L'édition 2003 de cette « feuille de route », l'ITRS 2003 est disponible depuis peu [6]. Elle se divise en de nombreux chapitres dont la conception, le test, les procédés d'intégration concernent plus particulièrement le concepteur de systèmes sur une puce.

Dans le volet conception, l'ITRS annonce qu'à long terme, la conception de systèmes devrait se focaliser sur la communication entre les blocs. Aux niveaux d'intégration prévus, la robustesse vis-à-vis des variations technologiques, le contrôle de la consommation d'énergie et la synchronisation entre les blocs devraient impliquer progressivement le recours aux techniques de conception de circuits numériques asynchrones.

Problématique

Il parait donc légitime de s'intéresser aux circuits numériques asynchrones dans un cadre de recherche pour préparer l'avenir de la conception (spécification, modélisation, simulation, vérification, synthèse, test) des circuits numériques [7].

De nombreux types de circuits numériques asynchrones ont été développés depuis les années 1950. Ils reposent sur différents types de délais, différents types de couplage entre données et signalisation et différents types de codage des données. En particulier, chaque hypothèse de délai conduit à un flot de conception ad hoc. Aujourd'hui, il existe donc de nombreuses solutions permettant de concevoir des circuits numériques asynchrones.

Cependant, les circuits numériques synchrones demeurent prépondérants. Suivant les recommandations de l'ITRS, les techniques de conception de circuits numériques asynchrones doivent progressivement diffuser dans le monde des circuits numériques synchrones. Il paraît raisonnable de se placer dans le cadre d'une approche mixte synchrone/asynchrone pour en homogénéiser la conception. Il faut alors prendre en compte les spécificités des circuits numériques asynchrones et les intégrer dans un environnement de conception de circuits numériques synchrones. Ensuite, il faut pouvoir modéliser la plupart des circuits numériques asynchrones, sinon tous dans cet environnement.

Cela implique une étude théorique des circuits numériques asynchrones. Nous sommes conduit à considérer les systèmes à événements discrets dont les circuits numériques asynchrones sont une sous-classe. Dans l'hypothèse synchrone, on peut écrire un programme en faisant abstraction des délais. On peut voir ce processus d'abstraction des délais comme une conséquence du caractère borné des délais. Dans l'hypothèse asynchrone, les délais ne sont pas nécessairement bornés. Le processus d'abstraction des délais de la spécification ne va pas de soi.

Nous établissons que pour spécifier un circuit numérique asynchrone par un programme faisant abstraction des délais, il faut et il suffit que le circuit vérifie trois propriétés fondamentales, i.e. persistance, sûreté, vivacité, que nous appelons propriétés d'insensibilité aux délais. Nous démontrons ce théorème dans le cadre des réseaux de Petri ordinaires. Mais ce résultat ne devrait pas dépendre du cadre formel utilisé pour la démonstration. Corollairement, ce théorème fournit une définition de la correction de ces circuits. Elle permet

d'inscrire leur conception dans un flot descendant. C'est la partie théorique de la contribution de cette thèse.

Notre objectif se trouve donc précisé : il s'agit d'établir un modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais s'intégrant dans un standard de conception de systèmes numériques.

Cet objectif implique de choisir un environnement de conception de circuits numériques standard supportant la sémantique d'exécution des circuits numériques asynchrones. Or, un tel standard n'existe pas. Il nous faut alors choisir un standard de conception, le plus proche et le plus adapté possible. Il se trouve que le plus souvent le logiciel, d'un niveau d'abstraction plus élevé que le matériel se comporte de manière asynchrone. On se tourne par conséquent vers un environnement de conception de systèmes numériques. En effet, un système numérique se décompose en général en une partie matérielle et une partie logicielle. Un environnement supportant la conception de systèmes numériques doit supporter le modèle de calcul asynchrone que l'on retrouve au niveau logiciel. Un tel environnement doit également supporter le modèle de calcul synchrone à un niveau de granularité suffisamment fin pour représenter un circuit numérique au niveau transfert de registre et finalement au niveau porte logique.

Nous menons une étude théorique sur les systèmes à événements discrets afin de dégager une condition nécessaire et suffisante d'indépendance des délais pour la spécification des circuits numériques asynchrones. Nous passons en revue les différents types de circuits asynchrones. Nous choisissons un standard de conception de systèmes numériques.

Nous disposons alors de tous les éléments pour proposer un modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais s'intégrant dans un standard de conception de systèmes numériques. Ce modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais s'intégrant dans un standard de conception de systèmes numériques représente la partie pratique de la contribution de cette thèse.

Contribution

La contribution de cette thèse se divise donc en deux parties : une partie théorique et une partie pratique.

Sur le plan théorique, nous établissons le résultat dont l'énoncé suit.

Théorème : pour spécifier un circuit numérique asynchrone par un programme faisant abstraction des délais, il faut et il suffit que le circuit vérifie trois propriétés fondamentales, i.e. persistance, sûreté et vivacité, que nous appelons propriétés d'insensibilité aux délais.

Nous démontrons ce théorème dans le cadre des réseaux de Petri ordinaires. Mais ce résultat ne devrait pas dépendre du cadre formel utilisé pour la démonstration. Cette caractérisation par les propriétés de persistance, sûreté et vivacité que nous appelons encore propriétés d'insensibilité aux délais fournit une définition de la correction permettant d'inscrire la conception de ces circuits dans un flot descendant de conception.

Sur le plan pratique, nous établissons le premier modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais s'intégrant dans un standard de conception de systèmes numériques, nommément le standard SystemC v2.0.1.

Etude

Cette contribution découle de l'étude suivante décomposée en 5 chapitres.

Au chapitre I, nous étudions les systèmes à événements discrets. Nous établissons une condition nécessaire et suffisante pour spécifier un circuit numérique asynchrone indépendamment des délais. C'est la partie théorique de la contribution de cette thèse. Elle nous permet d'inscrire la conception des circuits numériques asynchrones dans un flot descendant de conception et de définir un critère de correction pour un tel système.

Au chapitre II, nous passons en revue les différents types de circuits asynchrones afin d'en dégager les caractéristiques principales, notamment au niveau des modèles de délais.

Au chapitre III, nous présentons SystemC v2.0.1, standard de conception de systèmes numériques, standard vers lequel nous nous tournons après avoir passé en revue différentes possibilités. Nous insistons sur deux points : la structure et le noyau de simulation.

Les deux derniers chapitres décrivent la partie pratique de la contribution de cette thèse. Ils sont fortement dépendants du chapitre III. Nous procédons de manière incrémentale pour intégrer progressivement les différents concepts du modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais s'intégrant dans un standard de conception de systèmes numériques.

Au chapitre IV, nous adaptons d'abord la sémantique de SystemC v2.0.1 aux circuits numériques asynchrones après avoir soigneusement étudié les limitations du standard. Puis, nous établissons une structure type de circuit dans SystemC v2.0.1 basée sur le paradigme original de programmation de SystemC v2.0.1 permettant de séparer calcul et communication. Ensuite, nous focalisons sur les circuits quasi insensibles aux délais. Nous les formalisons en logique booléenne linéaire temporelle (LTL) du premier ordre. Nous appliquons le principe de modélisation basé sur une structure type en SystemC v2.0.1 aux éléments des circuits numériques asynchrones formalisés en LTL. Nous complétons le modèle obtenu par les éléments producteur et consommateur chargés de l'initialisation de la simulation et de la modélisation de l'environnement du circuit. Il faut effectivement garder à l'esprit l'absence d'horloge globale cadençant l'évolution du circuit. Nous étudions la temporisation du modèle. Cette temporisation doit être cohérente avec le comportement du système avant et après prise en compte des délais. Enfin nous mettons en oeuvre le modèle obtenu sur un circuit de taille réduite, mais significatif. La contribution de ce chapitre est un modèle de circuits numériques asynchrones, s'intégrant dans le standard SystemC v2.0.1 de conception de systèmes numériques.

Au chapitre V, nous formalisons en LTL les propriétés d'insensibilité aux délais, i.e. persistance, sûreté et vivacité des circuits numériques asynchrones quasi insensibles aux délais. A partir de là, nous instrumentons le modèle de circuits numériques asynchrones pour la vérification des propriétés d'insensibilité aux délais. Puis nous étudions la composition de cette instrumentation avec la temporisation du modèle. Enfin nous mettons en œuvre le modèle obtenu sur plusieurs circuits de taille réduite, mais significatifs. La contribution de ce chapitre est un modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais, s'intégrant dans le standard SystemC v2.0.1 de conception de systèmes numériques.

En conclusion, notre objectif étant atteint, on présente d'une part la partie théorique de la contribution de cette thèse et d'autre part, la partie pratique de la contribution de cette thèse : le premier modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais, s'intégrant dans un standard de conception de systèmes numériques, dans son ensemble. Puis on élargit la perspective suivant trois directions

principales : l'évolution du standard SystemC v2.0.1, l'amélioration du modèle de circuits numériques asynchrones lui-même et la complétion du flot de conception.

I Systèmes à événements discrets

L'objectif de ce chapitre est l'étude théorique des systèmes à événements discrets dont les circuits numériques asynchrones sont une sous-classe. Nous établissons une condition nécessaire et suffisante pour spécifier un circuit numérique asynchrone indépendamment des délais. Voilà la partie théorique de la contribution de cette thèse. Elle nous permet d'inscrire la conception des circuits numériques asynchrones dans un flot descendant de conception. Corollairement, cela définit un critère de correction pour un tel système.

1 Classification des systèmes

Les caractéristiques de la technologie d'intégration d'un système se découpent en deux grandes familles, l'une physique et l'autre architecturale.

Au niveau *physique*, tout, i.e. tension d'alimentation, température, caractéristiques statiques et dynamiques des dispositifs intégrés, se ramène à la nature des signaux véhiculés dans le système. Ces signaux sont analogiques (continus), ce qui est en dehors de notre propos, ou bien numériques (discrétisés).

Au niveau *architectural*, tout, i.e. les composants de base de la conception, se ramène à la nature du modèle de temps considéré. Le temps peut être vu comme continu (analogique), ce qui est en dehors de notre propos, ou bien discret (logique) et dans ce dernier cas une architecture peut être *asynchrone* (deux occurrences d'événement peuvent être aussi proche l'une de l'autre dans le temps sans se confondre) ou bien *synchrone* (la simultanéité est introduite par un subtil passage à la limite).

On retrouve la classification habituelle des systèmes selon deux axes, un axe des temps et un axe des valeurs, comme représentée sur la Figure 3. On s'intéressera plus particulièrement à la classe des systèmes à événements discrets pour lesquels le temps est discret et les signaux numériques. Dans ce contexte, les événements sont atomiques, distinguables et de durée nulle.

Les systèmes à événements discrets se divisent eux en deux branches : les systèmes à événements discrets asynchrones et les systèmes à événements discrets synchrones qu'on présentera progressivement dans la suite.

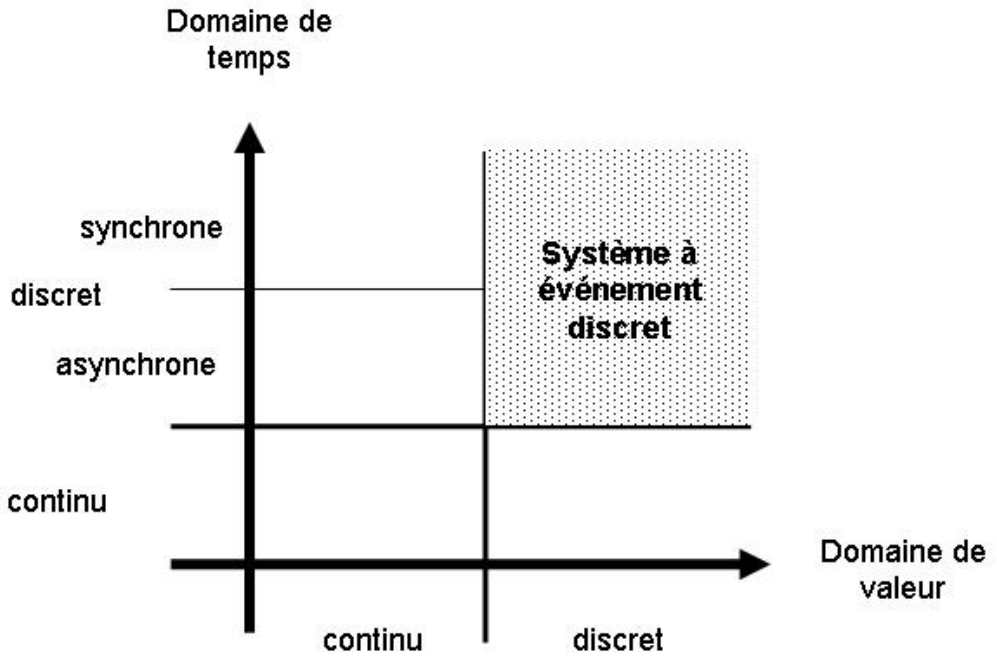


Figure 3 : Diagramme de classification des systèmes

On considère qu'un tel système est constitué de deux parties communicant : une machine, l'objet de l'étude, et son environnement d'exécution, le tout formant un système fermé tel que sur la Figure 4. En fait, pour affiner la description du système, on peut dire que l'environnement d'exécution est constitué de plusieurs parties. La machine associée au générateur de stimuli et aux sondes chargées d'observer l'évolution de la machine forme le dispositif de simulation. Le dispositif de simulation associé au simulateur forme le banc de simulation.

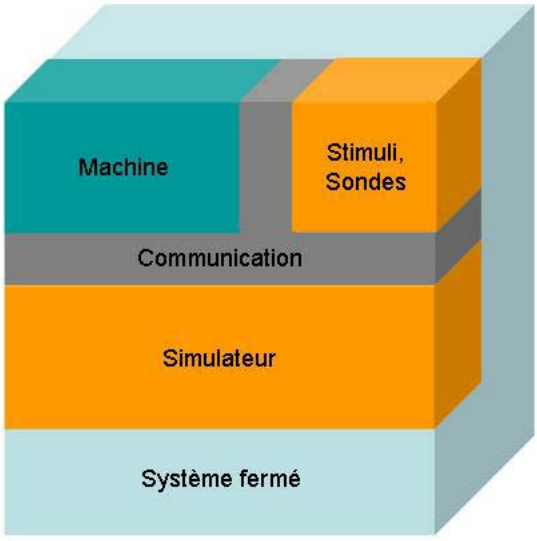


Figure 4 : Système fermé composé d'une machine et de son environnement d'exécution

2 Cadres formels

Il existe de nombreux cadres formels pour la spécification et la modélisation des systèmes à événements discrets. On pourra classer ces théories suivant deux approches complémentaires : l'approche observationnelle et l'approche structurelle.

L'approche observationnelle, comme son nom l'indique se focalise sur la description du comportement d'un système. A partir de ce comportement, on peut extraire une structure du système. La théorie des traces [40] d'Antoni Mazurkiewicz et CCS (« a Calculus of Communicating Systems » en anglais) [41] de Robert Milner en sont deux exemples.

L'approche structurelle, elle, se focalise sur la description de la structure des éléments d'un système. Ces structures peuvent être composées pour définir un système. Le comportement du système dérive alors de la définition de ces structures. On pourra citer CSP (« Communicating Sequential Processes » en anglais) [39] de Tony Hoare et les réseaux de Carl Petri [28] en exemple.

Ces approches sont naturellement liées. Dans CSP, la théorie des traces permet de définir mathématiquement la sémantique des différentes opérations sur les processus. Eike Best [42] utilise CSP pour définir la sémantique dynamique des réseaux de Petri. Ursula Goltz [43] étudie le lien entre CCS et réseaux de Petri.

Une étude comparative détaillée de ces cadres formels sort largement du cadre de cette thèse. On traite en annexe (voir A) un exemple de système à événements discrets décrit à l'aide de CSP. On calcule dans ce cadre la trace d'exécution du système dans deux cas de figure : le cas de la composition parallèle déterministe et le cas de la composition parallèle non déterministe. Cet exemple illustre les notions de processus, de composition et de trace. Il faut noter que ce développement ne tient pas compte de la durée des événements, ni du type d'implémentation, synchrone ou asynchrone. Pourtant, il demeure complexe malgré la taille réduite de la spécification étudiée. On fournit également les réseaux de Petri ordinaires correspondants.

3 Machine séquentielle

3.1 Introduction

Le modèle de calcul le plus simple et le plus courant pour une machine séquentielle est celui de la machine d'états finis. Le théorème de Kleene nous indique que ce modèle, dans sa variante non déterministe équivaut à un langage régulier. L'algorithme de Thompson permet de construire le diagramme d'état correspondant à son expression régulière. La construction des sous-ensembles permet de transformer une machine non déterministe en machine déterministe [27]. La Figure 5 fournit un exemple d'expression régulière, la machine d'état non déterministe obtenue par l'algorithme de Thomson et la machine d'états finis déterministe obtenue par construction des sous-ensembles.

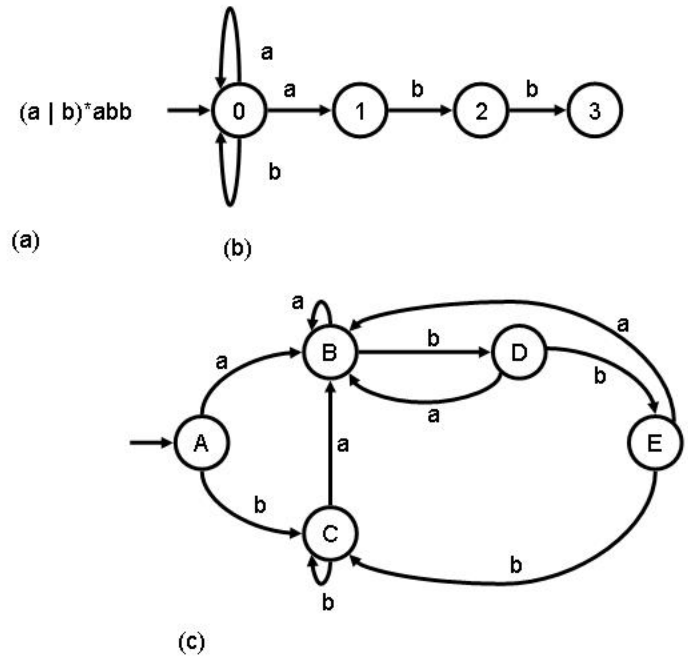


Figure 5 : (a) Expression régulière. (b) Machine d'états finis non déterministe. (c) Machine d'états finis déterministe

3.2 Syntaxe

Une machine d'états finis est définie mathématiquement par un septuplet $(I, O, S, s_i, S_f, \delta, \lambda)$ où :

- I est un ensemble fini et non vide appelé alphabet d'entrée,
- O est l'alphabet de sortie,
- S est l'ensemble fini et non vide des états,
- $s_i \subseteq S$ est l'état initial,
- $S_f \subseteq S$ est l'ensemble des états finaux,
- $\delta : S \times I \rightarrow S$ est la fonction de calcul de l'état suivant,
- $\lambda : S \times I \rightarrow O$ est la fonction de calcul des sorties pour une machine de Mealy (ou bien $\lambda : S \rightarrow O$ pour une machine de Moore).

3.3 Sémantique

Le calcul débute dans l'état initial à l'instant initial et les symboles d'entrée sont présentés successivement à l'entrée de la machine par l'environnement. La machine évolue d'après la fonction de calcul de l'état suivant et fournit un symbole de sortie à l'environnement.

Si l'on se penche de plus près sur cette sémantique d'exécution, on constate certaines imprécisions. En effet, que se passe-t-il si les fonctions δ et λ terminent au bout d'un temps indéterminé ou bien terminent en même temps qu'arrive un nouveau symbole d'entrée ou encore si un nouveau symbole d'entrée est présenté et interprété par la machine alors que le symbole de sortie n'est pas encore calculé?

4 Machine parallèle

4.1 Introduction

Pour raisonner sur les machines parallèles dont les machines séquentielles sont un cas particulier, on s'appuie maintenant sur les réseaux de Petri [28]. Ce modèle mathématique a été introduit au début des années 1960 par Carl Petri dont l'objectif initial était de construire un modèle de calcul universel. Il a bien sûr échoué. C'est une conséquence du théorème de Gödel. En effet, toute théorie suffisamment sophistiquée est incomplète. Mais les réseaux de Petri ont fait l'objet de plus de cinq mille publications depuis lors. On pourra se tourner vers [29][30][31][32][33] pour des présentations synthétiques concernant leur modélisation, leurs propriétés et les problèmes de complexité associés à leur vérification. Ces réseaux se déclinent en de très nombreuses variantes [35], sans doute pour remplir l'objectif initial d'universalité que s'était fixé Carl Petri.

4.2 Syntaxe

Un réseau de Petri ordinaire N associé au marquage initial M_0 est un quadruplet (P, T, F, M_0) noté (N, M_0) où :

- P est un ensemble fini de place,
- T est un ensemble fini de transitions,
- $F \subseteq (P \times T) \cup (T \times P)$ est un ensemble d'arcs,
- $M_0 : P \rightarrow \{0, 1, 2, \ldots\}$ est le marquage initial,
- $P \cap T = \varnothing$ et $P \cup T \neq \varnothing$.

Le marquage initial définit le nombre de jetons, potentiellement infini qu'une place peut contenir à l'instant initial. Une transition *t* possède des places d'entrée et des places de sortie. Les places d'entrée de *t* sont celles reliées à *t* par un arc de $(P \times T)$. Les places de sortie de *t* sont celles reliées à *t* par un arc de $(T \times P)$. De même, une place *p* possède des transitions d'entrée et des transitions de sortie. Les transitions d'entrée de *p* sont celles reliées à *p* par un arc de $(T \times P)$. Les transitions de sortie de *p* sont celles reliées à *p* par un arc de $(P \times T)$.

La Figure 6 montre la plupart des motifs que cette syntaxe graphique permet d'exprimer : choix, convergence, synchronisation et fourche.

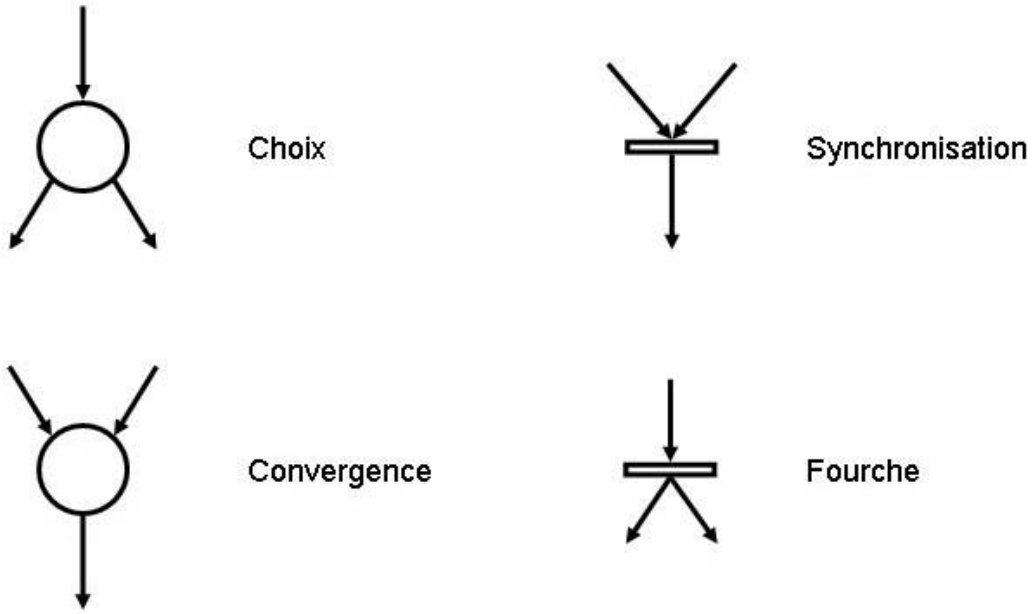


Figure 6 : Motifs de base des réseaux de Petri ordinaires

4.3 Sémantique

Le calcul démarre à l'instant initial avec le réseau configuré par le marquage initial. Une transition est dite sensibilisée si chaque place d'entrée est marquée par au moins un jeton. Une transition sensibilisée peut être tirée ou pas. En effet, à un instant donné plusieurs transitions peuvent être sensibilisées simultanément. Dans le mode de fonctionnement asynchrone, la simultanéité n'existe pas. L'environnement choisit par conséquent de tirer aléatoirement l'une ou l'autre des transitions sensibilisées. Dans le mode synchrone, toutes les transitions sont tirées simultanément. Le tir d'une transition supprime un jeton de chaque place d'entrée et ajoute un jeton à chaque place de sortie.

4.4 Délais

Les délais peuvent être spécifiés de manière non probabiliste par une valeur ou un intervalle ou encore de manière stochastique par une densité de probabilité. Ils sont traditionnellement associés soit aux places, soit aux transitions. C'est équivalent. On choisit d'associer les délais aux transitions. Les places d'entrée d'une transition sont alors vues comme les préconditions de l'exécution d'une tâche, les places de sortie comme ses post conditions.

5 Propriétés

5.1 Introduction

Plusieurs notions propres aux systèmes à événements discrets sont à mentionner. Elles trouvent une définition élégante et claire dans le contexte des réseaux de Petri ordinaires. En particulier, l'accessibilité, la sûreté, la persistance et la vivacité sont présentées dans les paragraphes suivants.

5.2 Accessibilité

L'accessibilité est une propriété en relation avec le graphe d'états du système ainsi qu'avec le langage qu'il exprime. Sa construction conduit à une explosion combinatoire due à tous les entrelacements des actions à considérer. On parle aussi de simulation exhaustive pour la construction de ce graphe. Le tir d'une transition sensibilisée va changer le marquage conformément à la sémantique d'exécution du réseau de Petri ordinaire. Une séquence de tirs va produire une séquence de marquages. Un marquage M_n est dit accessible depuis le marquage M_0 s'il existe une séquence de tirs qui mène de M_0 à M_n. L'ensemble de tous les marquages accessibles depuis M_0 pour N est noté $R(N, M_0)$. L'ensemble de toutes les séquences de tirs possibles depuis M_0 pour N est notée $L(N, M_0)$.

5.3 Persistance

La persistance est une propriété liée à la dynamique du système. Une violation de la persistance se produit lorsque le tir d'une transition désactive une transition déjà sensibilisée. Ceci produit un aléa de fonctionnement. En effet, une transition est un événement atomique. Dès lors qu'une transition est sensibilisée, elle doit être tirée à un moment donné. Sa désensibilisation ou encore son interruption provoque une violation de l'atomicité de l'action considérée. Un réseau de Petri ordinaire (N, M_0) est dit persistant si pour toute paire de transitions sensibilisées, le tir de l'une n'entraîne pas la désensibilisation de l'autre.

5.4 Sûreté

La sûreté est une propriété liée à la dynamique du système et plus précisément à l'accès à une même place à travers des chemins différents. Elle garantit l'absence de point d'accumulation de jetons dans le réseau. Un réseau de Petri ordinaire (N, M_0) est dit sûr si pour tout marquage M accessible depuis M_0, chaque place de M contient au plus un jeton et un seul.

5.5 Vivacité

La vivacité est une propriété héritée du monde du logiciel. Elle caractérise l'absence de blocage dans un système d'exploitation. Elle peut également être vue comme une garantie de bon fonctionnement de la communication entre la machine et son environnement. Une transition *t* est dite vivace si pour tout marquage M de $R(N, M_0)$, *t* peut être tirée au moins une fois dans une séquence de tirs de $L(N, M)$. Par extension, un réseau de Petri ordinaire est dit vivace si toutes ses transitions le sont aussi.

5.6 Puissance d'expression, décidabilité et complexité

Un réseau de Petri ordinaire ne possède pas la puissance d'expression de la machine de Turing [44]. Il faut recourir aux réseaux de Petri dits contextuels (adjonction d'un arc inhibiteur permettant de tester l'absence d'un jeton dans un place) pour accéder à cette puissance d'expression.

La vérification des propriétés d'accessibilité, de persistance, de sûreté et de vivacité est un problème décidable dans le cadre des réseaux de Petri ordinaires [31].

Par contre, dans le cas des réseaux de Petri contextuels, c'est un problème indécidable. Cependant, il existe une condition suffisante pour garantir la décidabilité de ces propriétés. Il suffit que les places d'entrée des arcs inhibiteurs soient sûres [33].

Dans le cas général, ce problème est de complexité PSPACE, i.e. de taille polynomiale pour l'espace mémoire et à temps non borné pour l'algorithme de vérification [31].

6 Abstraction des délais

6.1 Approche synchrone

L'approche synchrone considère que les délais sont bornés. Il existe donc une limite supérieure aux délais de fonctionnement du système. Cette limite supérieure permet d'introduire une horloge commune à la machine et à son environnement. Cette horloge cadence le fonctionnement de tout le système. Ainsi, la vitesse de fonctionnement du système est limitée par sa composante la plus lente. Sur le plan pratique, on prend une marge suffisamment grande sur le temps de cycle pour la période d'horloge afin de limiter les problèmes de métastabilité, i.e. d'indétermination, induits par l'échantillonnage des signaux et les variations des paramètres technologiques. Cette approche est donc souvent complétée par des mécanismes de tolérance aux fautes. Traditionnellement, l'ajout d'un bit de parité au bus de données suffit sur le plan matériel pour la détection des erreurs et des mécanismes de correction logicielle sont prévus dans les protocoles de communication comme spécifié par exemple au niveau « liaison » du standard réseau à sept couches OSI (« Open Systems Interconnection » en anglais) [8].

Les opérations sont séquencées de la façon suivante :

- la machine démarre à l'instant initial dans l'état initial avec la sortie prête,
- l'environnement présente un symbole d'entrée avant un coup d'horloge,
- tout se passe comme si les calculs de l'état suivant et du symbole de sortie s'effectuaient à temps nul en étant déclenchés par le coup d'horloge,
- l'environnement reçoit un symbole de sortie après le coup d'horloge.

Le troisième point correspond au concept de simultanéité, apanage de l'approche synchrone. Toutes les actions sont alignées sur le coup d'horloge. Il n'y a plus qu'un seul événement qui caractérise le pas de calcul, le coup d'horloge.

6.2 Approche asynchrone

Dans l'approche asynchrone, la simultanéité n'existe pas. Les notions de limite supérieure sur laquelle s'aligneraient les événements et donc d'horloge globale n'ont pas de sens. Il faut donc recourir à un autre stratagème pour abstraire les délais de la spécification et rendre la programmation de haut niveau possible. Nous allons nous appuyer sur les propriétés de persistance, sûreté et vivacité pour définir le concept d'insensibilité aux délais et ainsi obtenir une condition nécessaire et suffisante d'abstraction des délais d'un réseau de Petri ordinaire. On trouve un raisonnement proche chez David Misunas [36] dans le cadre des circuits indépendants de la vitesse définis au chapitre II.

Considérons donc un réseau de Petri ordinaire (N, M_0) auquel on associe un délai positif ou nul, fini et non borné à chaque transition ; réseau qu'on nommera indifféremment spécification.

6.2.1 Théorème d'abstraction des délais dans les systèmes à événements discrets asynchrones

6.2.1.1 Condition nécessaire

Il est nécessaire qu'un réseau de Petri soit persistant, sûr et vivace pour être insensible aux délais.

Supposons que le réseau ne soit pas persistant. Alors, il existe une paire de transitions (T_0, T_1) telle que le tir de l'une désactive l'autre, comme représenté sur la Figure 7.

Si l'on tire les transitions T_0 et T_1 en parallèle à partir du marquage (a), alors suivant que T_0 est plus rapide que T_1, le réseau évoluera vers le marquage (b) plutôt que vers le marquage (c). Non seulement le fonctionnement spécifié dépend de la distribution des délais dans le réseau (et ce quelque soit l'ordre d'exécution des transitions choisi par l'environnement), mais les actions ne peuvent plus être considérées comme atomiques.

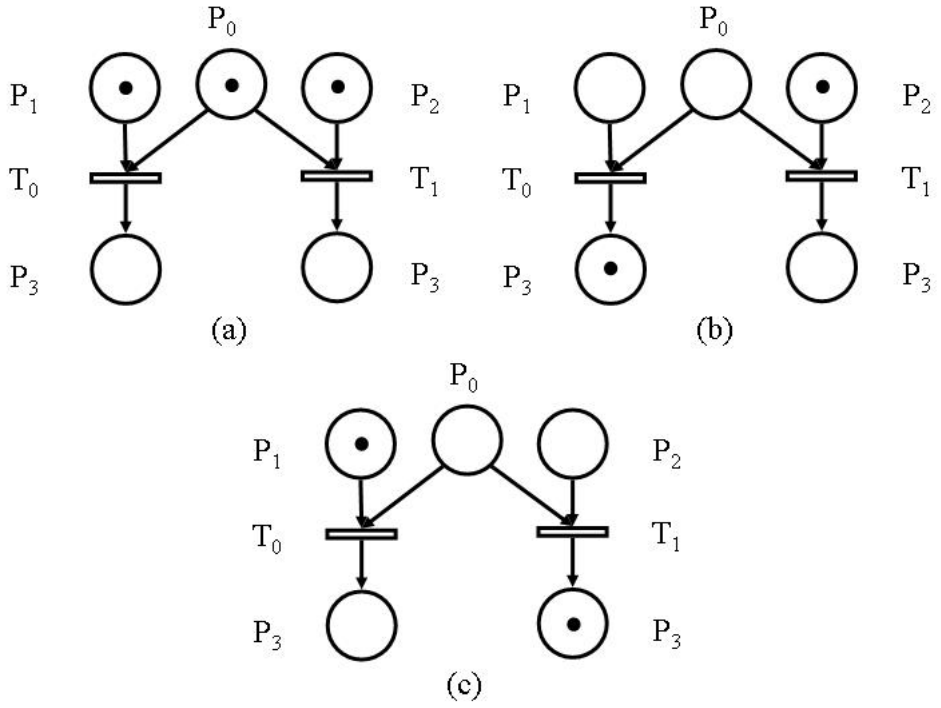


Figure 7 : Schéma d'exécution d'un réseau de Petri non persistant

Supposons que le réseau ne soit pas sûr comme sur la Figure 8. Alors, il existe un marquage du réseau tel qu'une place contienne au moins deux jetons tel qu'en (a). Supposons que l'on tire une première fois la transition T_0. Au bout d'un certain temps on obtient le marquage (b) pour lequel T_0 et T_1 sont sensibilisés. Supposons qu'on débute concurremment le tir des transitions T_0 et T_1. Si le délai associé à la transition T_0 est beaucoup plus court que celui associé à la transition T_1, on obtient le marquage (c). Dans le cas contraire, on obtient le marquage (d). Le fonctionnement spécifié dépend donc de la distribution des délais dans le réseau.

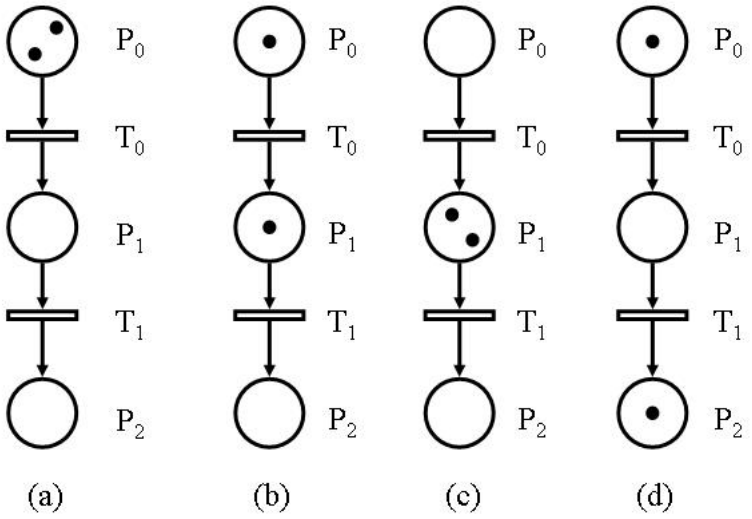


Figure 8 : Exemple de fonctionnement d'un réseau de Petri non sûr

Supposons que le réseau ne soit pas vivace. Soit il existe un marquage atteignable considéré comme correct pour lequel plus aucune transition n'est sensibilisée et l'exécution est terminée. Il suffit alors de réinitialiser la machine dès qu'on a observé cet état pour se ramener à un comportement vivace. Soit le système est bloqué dans un état indésirable. Soit le système est bloqué dans un état pendant trop longtemps. Et le fonctionnement est incorrect.

6.2.1.2 Condition suffisante

Il est suffisant qu'un réseau de Petri soit persistant, sûr et vivace pour être insensible aux délais.

Raisonnons par l'absurde sur la contraposée. Si la spécification n'est pas insensible aux délais, on peut considérer deux niveaux de spécifications. Le premier niveau de spécification n'inclut pas la spécification des délais. Les transitions sont instantanées. Le comportement spécifié est considéré comme correct. C'est la spécification de référence. Le second niveau inclut les délais. Suivant la répartition des délais sur les transitions, son comportement n'est pas conforme à la spécification de référence. On choisit arbitrairement une configuration de délais telle que le réseau possède au moins un marquage non permis par rapport à la spécification de référence. Ainsi, soit ce marquage erroné possède au moins un jeton en plus, soit il possède au moins un jeton en moins par rapport au marquage attendu.

Supposons que le réseau de référence soit persistant, sûr et vivace. Alors, d'après le théorème de Michel Hack [38], ce réseau est couvert par un ensemble de machines d'états finis fortement connexes possédant chacune un jeton au marquage initial. Si l'une de ces machines contient un jeton en plus, alors il existe une séquence de tirs qui viole la propriété de sûreté. Si l'une d'elle contient un jeton en moins, alors elle perd de fait sa propriété de vivacité. On aboutit à une contradiction.

7 Conclusion et contribution

Dans le cadre des systèmes à événements discrets, les actions sont atomiques, distinguables et de durée nulle. On peut classer ces systèmes suivant deux catégories : les systèmes à événements discrets synchrones et les systèmes à événements discrets asynchrones. Dans le premier cas, on considère que la simultanéité des actions existe, alors que dans le second cas, la simultanéité des actions n'existe pas. Les circuits numériques asynchrones sont une sous-classe des systèmes à événements discrets asynchrones.

Dans le cas synchrone, il existe une borne supérieure pour les délais d'exécution des actions. C'est ce qui permet d'en faire abstraction dans le cadre d'un langage de spécification.

Dans le cas asynchrone, une telle borne supérieure n'existe pas nécessairement. On montre alors qu'un réseau de Petri ordinaire est une spécification insensible aux délais d'un système à événements discrets asynchrone si et seulement si, il vérifie les propriétés de persistance, de sûreté et de vivacité, que nous nommons propriétés d'insensibilité aux délais.

Cette condition nécessaire et suffisante permet de spécifier des systèmes à événements discrets asynchrones sans se préoccuper des délais. Ces propriétés fournissent corollairement une définition de la correction du système en l'absence de spécification de référence.

La contribution de ce chapitre réside dans ce théorème : un réseau de Petri ordinaire est une spécification insensible aux délais d'un système à événements discrets asynchrone si et seulement si il vérifie les propriétés d'insensibilité aux délais, i.e. persistance, sûreté et vivacité.

II Les modèles existants de circuits numériques asynchrones

L'objectif de ce chapitre est d'établir un état de l'art dans le domaine des circuits numériques asynchrones. Nous en dégageons les principales caractéristiques, notamment la modélisation des délais.

1 Introduction

Comme il a été évoqué au chapitre I, un système est fortement conditionné par le modèle de temps sur lequel il s'appuie. Il en va de même pour les systèmes à événements discrets asynchrones et donc pour les circuits numériques asynchrones. Ces circuits sont constitués d'opérateurs logiques interconnectés par des fils. Dans ce contexte, quatre modèles de temps se distinguent. Ils sont présentés du moins restrictif au plus contraignant :

- le modèle insensible aux délais présenté au paragraphe 2,
- le modèle quasi insensible aux délais présenté au paragraphe 3,
- le modèle indépendant de la vitesse présenté au paragraphe 4,
- le modèle fondamental présenté aux paragraphes 5 et 6.

Dans chacun de ces modèles, les délais associés aux opérateurs et aux fils sont finis et positifs ou nuls.

Dans le modèle insensible aux délais, les délais associés aux opérateurs et aux fils sont non bornés. Dans le modèle quasi insensible aux délais, les délais sont non bornés mais certaines fourches sont isochrones, i.e. le temps de propagation est identique dans chacune des branches de la fourche. Dans le modèle indépendant de la vitesse, les délais associés aux opérateurs sont non bornés et les délais associés aux fils sont nuls. Dans le modèle fondamental, les délais associés aux opérateurs et aux fils sont bornés.

Les circuits de Suhas Patil [37] et les circuits de Jo Ebergen [66] s'appuient sur le modèle de temps insensible aux délais. Les circuits de Alain Martin [17] s'appuient sur le modèle de temps quasi insensible aux délais. Les circuits de David Muller [11] s'appuient sur le modèle de temps insensible à la vitesse. Les circuits de Ivan Sutherland [19] et les circuits de David Huffman [21] s'appuient sur le modèle de temps fondamental.

Chacun de ces circuits ayant la particularité de fonctionner sans horloge globale, on parle aussi de circuit endochrone [12]. L'ordonnancement des actions et l'échange des données sont assurés par un mécanisme local de communication entre les opérateurs du circuit. Ce mécanisme local de communication varie en fonction des circuits et peut être caractérisé par deux critères : le protocole de communication et l'encodage des données (voir Figure 9).

Le protocole de communication peut être avec ou sans retour à zéro. On parle également de protocole quatre phases ou deux phases respectivement. Dans le premier cas, les signaux sont actifs sur niveau, alors que dans le deuxième les signaux sont actifs sur front.

Les données peuvent être codées séparément ou conjointement aux signaux de synchronisation. On parle respectivement de données groupées ou de code insensible aux délais. Dans le premier cas, le contrôle est séparé des données et les données sont

généralement codées en binaire. Dans l'autre cas, le contrôle et les données sont mêlés et on utilise souvent un code 1 parmi n, où n est un entier naturel déterminant la plage de représentation des nombres. La valeur de n est le plus souvent 2 ou 4 et on parle de code double rail ou de code quatre rails.

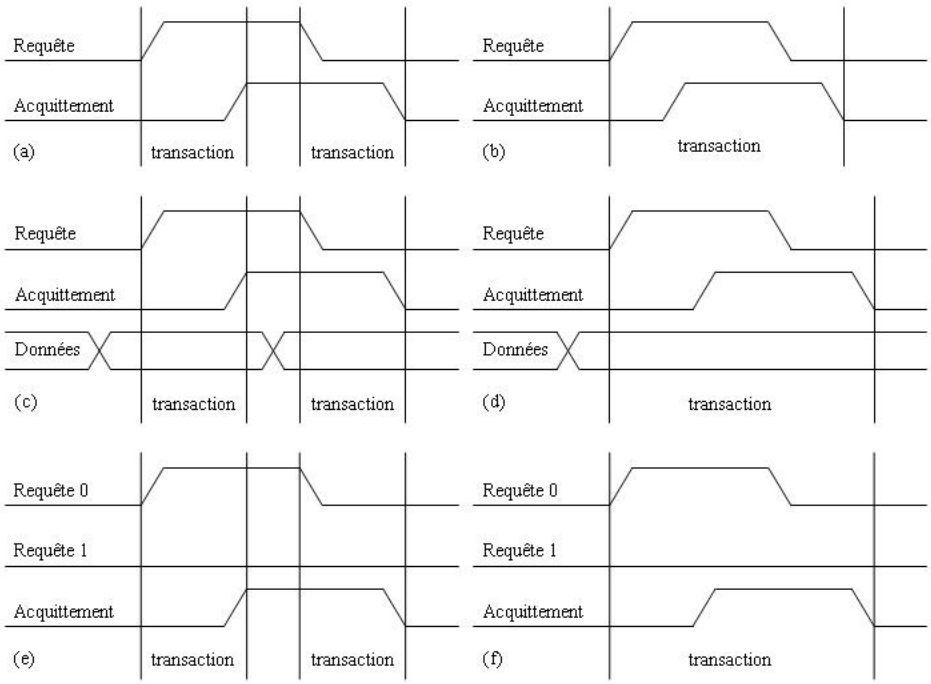


Figure 9 : Mécanismes de communication dans un circuit numérique asynchrone. (a) protocole NRZ, (b) protocole RZ, (c) données groupées NRZ, (d) données groupées RZ, (e) double rail NRZ, (f) double rail RZ

A ce niveau de granularité, le mécanisme de communication mis en jeu est sans mémoire. Ce mécanisme est bloquant dans la mesure où le protocole doit se dérouler complètement pour que le circuit puisse évoluer vers un nouvel état. Il existe aussi des mécanismes de communication à mémoire finie et à mémoire infinie correspondant à des niveaux d'abstraction plus élevés. Le mécanisme de communication peut alors être considéré comme bloquant ou non et la mémoire associée peut revêtir plusieurs formes (file, mémoire partagée, …).

Dans les sections suivantes, on trouve une présentation succincte des différents types de circuits numériques asynchrones. Les circuits quasi insensibles aux délais sont cependant développés plus largement car ils présentent de nombreux intérêts.

Bien que mettant en évidence les principaux types de circuits numériques asynchrones, cette présentation est loin d'être exhaustive. Pour en savoir plus sur les modèles de circuits numériques asynchrones et leur conception, on pourra se tourner vers les ouvrages [13][14] et l'article de synthèse [15] par exemple.

2 Les circuits insensibles aux délais

Nous nous appuyons sur l'approche basée sur les réseaux de Petri pour mettre en oeuvre sur le plan pratique, le modèle de circuits numériques asynchrones instrumenté pour la vérification

des propriétés d'insensibilité aux délais s'intégrant dans SystemC v2.0.1, standard de conception de systèmes numériques.

2.1 L'approche basée sur les réseaux de Petri

La classe des graphes marqués est une sous-classe des réseaux de Petri ordinaires qui trouve une traduction directe sous forme de circuit insensible aux délais [37]. Il s'agit des réseaux de Petri ordinaires desquels on a soustrait les choix et les convergences. Cependant, cette sous-classe possède un pouvoir d'expression très limité puisque ne sont traduisibles que les places, les places marquées, les fourches et les synchronisations. Des cellules sensibles aux délais sont ajoutées pour traduire la convergence et le choix et ainsi retrouver le pouvoir d'expression des réseaux de Petri ordinaires. La Figure 10 résume cette traduction directe.

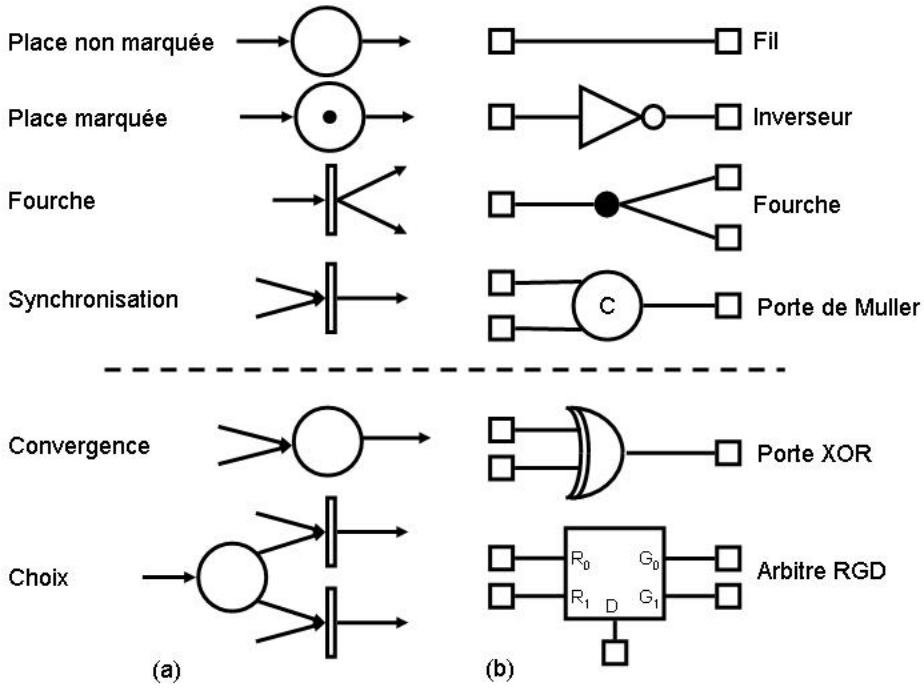


Figure 10 : Traduction directe des éléments d'un réseau de Petri ordinaire (a) vers un circuit logique (b)

2.2 L'approche basée sur les traces

L'insensibilité aux délais peut également être caractérisée en terme de traces [66]. Les traces d'un module insensible aux délais doivent respecter deux règles fondamentales :

- une trace ne peut contenir la succession de deux occurrences du même événement, i.e. la trace « aa » est interdite,
- un module doit être insensible à l'ordre d'arrivée des occurrences des événements d'entrée, i.e. si le module contient la trace « ab », il doit également contenir la trace « ba ».

Dans la deuxième règle, rien ne spécifie la distance temporelle entre deux événements en entrée. Cette approche nécessite donc la spécification de contraintes temporelles sur les entrées et donc ne peut être véritablement considérée comme insensible aux délais non plus.

Un certain nombre de ces modules peut ainsi être construit à partir de cette théorie (voir Figure 11). Dans une approche ascendante, on peut composer ces modules suivant un protocole de communication à base de poignées de main. Dans une approche descendante, il s'agit de décomposer une spécification en terme de traces pour faire apparaître ces modules et synthétiser le circuit à l'aide de cette bibliothèque.

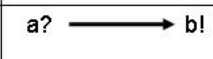
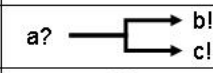
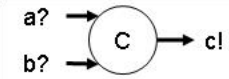
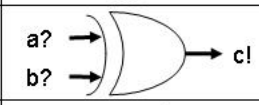
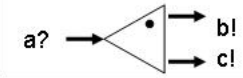
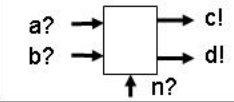
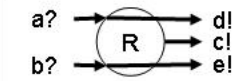
Nom	Spécification	Symbole
Wire	(a?;b!)*	
Fork	(a?;(b!||c!))*	
C-element	((a?||b?);c!)*	
XOR	((a?|b?);c!)*	
Toggle	(a?;b!;a?;c!)*	
Sequencer	(a?;c!)*||(b?;d!)*||(n?;(c!|d!))*	
RCEL	((a?;d!)²|(b?;e!)²|((a?;(d!||c!))²||(b?;(e!||c!))²))*	

Figure 11 : Table des principaux modules insensibles aux délais basés sur la théorie des traces

L'inconvénient de cette approche réside principalement dans la difficulté de spécifier un circuit complexe en termes de traces. Plusieurs approches ont permis de compenser cette limitation en élevant le niveau d'abstraction de la spécification au niveau d'une algèbre de processus ou d'un langage de programmation de plus haut niveau comme OCCAM [67], TANGRAM [68] ou BALSA [69].

3 Les circuits quasi insensibles aux délais

3.1 Introduction

Un modèle de circuits numériques asynchrones fidèle à la sémantique d'exécution du matériel en technologie CMOS a été établi par Alain Martin à l'Institut technologique de Californie autour de 1990. S'appuyant sur ce modèle, Alain Martin a caractérisé la classe des circuits numériques asynchrones insensibles aux délais par l'unicité de l'ensemble des successeurs d'une transition et a résolu la conjecture statuant que lorsqu'on cherche à implémenter un tel circuit à l'aide de portes à une seule sortie, seuls les fils, les inverseurs et les portes de Muller doivent être utilisés. Il a ensuite constaté que l'introduction de la contrainte temporelle dite de quasi insensibilité aux délais (on considère que certaines fourches sont isochrones) permet d'élargir la bibliothèque de cellules standard (à une seule sortie) permises [17]. Cette contrainte constitue l'hypothèse temporelle la plus faible que l'on puisse appliquer aux

circuits insensibles aux délais pour les implémenter à l'aide d'une bibliothèque de cellules standard.

Ce modèle est intéressant pour plusieurs raisons. Premièrement, comme on vient de l'évoquer, grâce à l'introduction du concept de fourche isochrone, la classe des circuits numériques asynchrones réalisables à l'aide de bibliothèques de composants logiques standard à une seule sortie est largement augmentée par rapport à la classe des circuits purement insensibles aux délais. Deuxièmement, ce modèle a la puissance d'expression de la machine de Turing [18]. On peut donc construire n'importe quel circuit logique à partir des éléments de ce modèle.

Nous présentons dans les sections suivantes ce modèle en partant des éléments de base vers les éléments les plus complexes en mettant en évidence les pré et post conditions à réaliser au cours de l'évaluation du modèle. Cela pose les définitions des circuits numériques asynchrones sur lesquelles nous nous appuyons par la suite pour construire le modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais s'intégrant dans SystemC v2.0.1, standard de conception de systèmes numériques.

3.2 Forme normale disjonctive

On note x le prédicat « x est vrai » et $\neg x$ le prédicat « x est faux ». Un littéral l est une variable booléenne x ou son complément à 2 noté $\neg x$. Un produit p est soit un littéral, soit la conjonction de plusieurs littéraux, notée $l_0 \wedge l_1 \wedge ...$ Une forme normale disjonctive est soit un produit, soit la somme de plusieurs produits, notée $p_0 \vee p_1 \vee ...$

3.3 Garde

Une garde est une expression booléenne sous forme normale disjonctive canonique.

3.4 Transition

Une affectation simple de la variable booléenne x à la valeur « vrai » ou à la valeur « faux » est notée $x\uparrow$ ou $x\downarrow$ respectivement. Une transition est l'exécution d'une affectation simple. Le résultat d'une transition de type $x\uparrow$ est la post condition x (« x est vrai »). Le résultat d'une transition de type $x\downarrow$ est la post condition $\neg x$ (« x est faux »).

3.5 Règle de production à une seule sortie

Une règle de production à une seule sortie est une implication entre une garde g et une variable booléenne x à faire transiter. Elle est notée $g \rightarrow x\uparrow$ et se lit « g implique x à vrai » ou $g \rightarrow x\downarrow$ et se lit « g implique x à faux » suivant la polarité de la transition.

L'exécution d'une règle de production est possible quand la garde associée est vraie. L'exécution d'une règle de production vise à établir le résultat de la transition cible. On considère que l'exécution d'une règle de production est correctement terminée quand le résultat de la transition cible est établi.

3.6 Porte à une seule sortie

Une porte à une seule sortie x est définie par la donnée de deux règles de production effectuant des transitions inverses sur cette sortie :

- $g_0 \rightarrow x\uparrow$

- $g_1 \rightarrow x\downarrow$

Si les deux gardes g0 et g1 sont complémentaires, on parle de porte statique. Sinon, il s'agit d'une porte dynamique nécessitant éventuellement l'introduction d'un point mémoire au niveau de la sortie. On voit d'emblée apparaître le problème de la contention sur la sortie dans le cas où les deux gardes seraient actives en même temps.

3.7 Règle de production à plusieurs sorties

Une règle de production à plusieurs sorties est une implication entre une garde g et une liste de variables booléennes $x_0, x_1, ...$ à faire transiter. Elle est notée $g \rightarrow x_0\uparrow, x_1\downarrow, ...$. Les transitions de la liste sont séparées par des virgules et sont exécutées en parallèle.

L'exécution d'une règle de production à plusieurs sorties est possible quand la garde associée est vraie. L'exécution d'une règle de production à plusieurs sorties vise à établir le résultat de toutes les transitions cibles. On considère que l'exécution d'une règle de production à plusieurs sorties est correctement terminée quand le résultat de chaque transition cible est établi.

3.8 Fourche

Dans ce modèle, une seule porte à plusieurs sorties est autorisée : la fourche. Cette porte comporte une seule variable d'entrée et deux variables de sortie composées en parallèle et séparées par une virgule :

- $g \rightarrow x\uparrow, y\uparrow$
- $\neg g \rightarrow x\downarrow, y\downarrow$

Dans une telle porte, toutes les transitions ont la polarité montante si la variable d'entrée n'est pas complémentée et la polarité descendante si la variable d'entrée est complémentée.

Si les durées des transitions montantes (respectivement descendantes) pour une même règle de production sont égales, on parle de fourche isochrone. C'est la contrainte de délai la plus faible s'appliquant à un circuit asynchrone purement insensible aux délais. Elle se retrouve également dans le modèle des circuits indépendants de la vitesse. Ce modèle fut introduit par David Muller [11] vers 1960 et ne tient pas compte des temps de propagation dans les fils d'interconnexion. A cette époque en effet, la technologie d'intégration était telle que les temps de propagation dans les fils étaient négligeables devant les temps de propagation dans les transistors. C'est le contraire aujourd'hui.

3.9 Circuit

Un circuit est un réseau d'opérateurs logiques ou portes. On suppose que l'état du circuit est entièrement caractérisé par les valeurs des entrées et des sorties de ces portes. Ces valeurs sont portées par des variables booléennes qui prennent une valeur donnée pour chaque état stable du circuit. On suppose que :

- le circuit est fermé, i.e chaque variable du circuit est à la fois l'entrée d'une porte et la sortie d'une autre porte,
- les variables changent de valeur de manière monotone,
- le temps de traversée dans les portes, de même que la durée des transitions est fini et positif ou nul, mais non nécessairement borné,

- à tout instant, toutes les règles de production d'un circuit sont exécutées en parallèle.

Lorsque l'exécution d'une règle de production intervient à partir d'un état du circuit dans lequel les résultats des transitions sont déjà établis, y compris ceux de la règle de production en cours d'exécution, on considère que l'exécution est ineffective. Ainsi le calcul effectué était redondant, puisque le calcul était déjà effectué. Lorsque l'exécution d'une règle de production intervient à partir d'un état du circuit dans lequel aucun des résultats des transitions de la règle en cours d'exécution n'est établi, on considère que l'exécution est effective.

3.10 Fonctionnement correct et fourche isochrone

On considère qu'un tel circuit fonctionne correctement si et seulement si il est persistant, vivace et sûr conformément au théorème du paragraphe I6.2. Il représente alors une spécification insensible aux délais d'un système à événements discrets asynchrone. On peut introduire la contrainte supplémentaire selon laquelle les temps de transition dans certaines fourches sont égaux. On obtient alors une spécification quasi insensible aux délais d'un système à événements discrets asynchrone dont la réalisation à l'aide d'une bibliothèque de composants logiques standard à une seule sortie devient possible.

4 Les circuits de Muller

Ces circuits furent introduits en 1959 par David Muller [11] et sont formalisés en terme d'états du circuit et de transitions d'un état à un autre. Dans ce modèle, une relation d'équivalence permet de caractériser un circuit indépendant de la vitesse comme un circuit dont toutes les séquences d'exécution aboutissent à la même classe d'équivalence terminale. Ces circuits peuvent être spécifiés de manière élégante à l'aide d'une variante des réseaux de Petri, les STG (« Signal Transition Graph » en anglais). Cette technique concerne donc plutôt la partie contrôle d'un circuit. Un algorithme en deux étapes permet d'une part de trouver un encodage binaire de l'espace des états en insérant automatiquement des états pour résoudre les conflits et d'autre part de synthétiser une logique sans aléa basée sur une bibliothèque standard de composants logiques à une seule sortie.

Petrify, logiciel développé à l'université de Catalogne permet de spécifier et de synthétiser ce type de circuits [16].

5 Les circuits de Sutherland

Ces circuits furent introduits par Ivan Sutherland en 1989 [19]. Ils ciblent un circuit complet, contrôle et chemin de données. Le contrôle est du type insensible aux délais. Des délais sont insérés pour synchroniser la partie contrôle et les registres du chemin de données (voir Figure 12). Ceci ramène le modèle de temps au mode fondamental. La synthèse de ce type de circuits est possible à partir d'un langage de description de haut niveau tel que Verilog [20] ou Balsa [69].

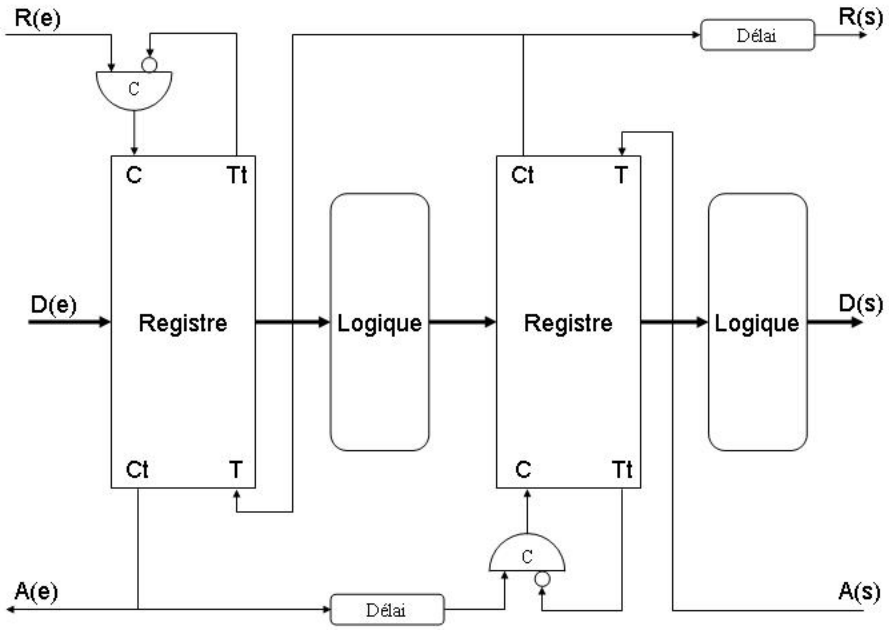


Figure 12 : Architecture d'un circuit micro pipeline élémentaire

6 Les circuits de Huffman

Ces circuits furent introduits en 1954 par David Huffman [21] et sont spécifiés sous forme d'une machine d'états finis asynchrone [22], d'une machine en mode rafale [23] ou bien d'une machine en mode rafale étendu [24]. Ils concernent donc plutôt la partie contrôle d'un circuit. En tant que variantes de la machine de Mealy, ce sont les plus proches des circuits synchrones. Il n'y pas d'horloge globale qui cadence l'évolution du circuit, mais on introduit des délais nécessaires à la stabilisation du calcul de l'état suivant et du symbole de sortie ainsi que des contraintes sur les entrées et les sorties du circuit (voir Figure 13). La synthèse de ces circuits se décompose en trois étapes comme pour les circuits synchrones : minimisation de l'espace des états, encodage binaire des états, optimisation logique. Il y a des contraintes à respecter sur les entrées qui conditionnent fortement l'algorithme de synthèse. La première est le mode SIC (« Single Input Change » en anglais) n'autorisant qu'un seul signal en entrée à changer. Trop restrictive, cette technique a ensuite évolué pour admettre le changement de plusieurs signaux en entrée ; c'est le mode MIC (« Multiple Input Change » en anglais). Il y a aussi des contraintes à respecter sur les sorties puisque la phase d'optimisation logique doit produire des circuits sans aléa.

Minimalist, outil développé à l'université de Columbia, New York, permet de spécifier et de synthétiser ce type de circuits [25].

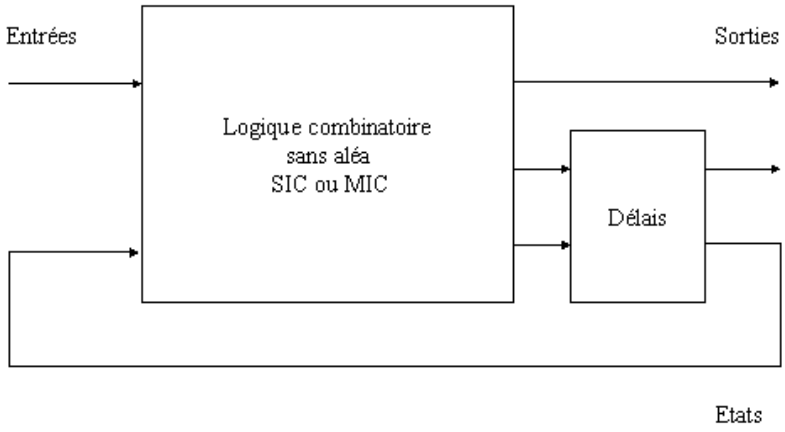


Figure 13 : Architecture d'un circuit de Huffman

III Le standard SystemC v2.0.1 de conception de systèmes numériques

L'objectif de ce chapitre est de présenter SystemC v2.0.1, standard de conception de systèmes numériques, langage vers lequel nous nous tournons après avoir passé en revue différentes possibilités. Nous insistons sur deux points fondamentaux pour la modélisation des systèmes à événements discrets en général, et asynchrones en particulier : la structure et le noyau de simulation.

1 Introduction

Si les réseaux de Petri représentent un modèle mathématique intéressant pour la modélisation des systèmes à événements discrets asynchrones (voir paragraphe I4), ils ne sont pas adaptés à la conception de systèmes sur une puce de grande taille pour plusieurs raisons.

Premièrement, il s'agit d'un langage graphique. La visualisation de petits exemples est très confortable et didactique, mais la conception d'exemples de grande taille nécessite une approche hiérarchique délicate à mettre en œuvre. Il s'agit en effet de graphes bipartites qu'il faut composer convenablement.

Deuxièmement, toujours en tant que langage graphique, ces réseaux se prêtent mieux à la description du flot de contrôle qu'à la description du flot de données. Notamment, la modélisation de l'affectation et de la comparaison de variables est connue pour être inutilement compliquée dans ce contexte.

Troisièmement, on s'est cantonné dans le chapitre précédent à la présentation d'un réseau de Petri ordinaire, qui ne possède pas la puissance d'expression de la machine de Turing [44]. Il faut recourir aux réseaux de Petri contextuels (adjonction d'un arc inhibiteur permettant de tester l'absence d'un jeton) pour accéder à cette puissance d'expression.

Enfin, ce n'est pas un langage standard.

On leur préfère donc un autre environnement pour la modélisation, la simulation et la vérification de systèmes à événements discrets d'envergure, de préférence non graphique, avec la puissance d'expression de la machine de Turing, si possible reposant sur un langage standard.

Intuitivement, le standard SystemC v2.0.1 de conception de systèmes numériques semble un bon candidat. SystemC v2.0.1 est avant tout, un langage de conception de systèmes numériques. Il n'est donc a priori pas adapté à la modélisation au niveau portes logiques [60]. Pourtant, il y a un exemple de modèle de bascule synchrone décrite au niveau transfert de registre dans le manuel utilisateur.

Quoi qu'il en soit, cette inadéquation apparente s'inscrit dans un contexte synchrone, où les langages de description de matériel tels que VHDL [9], Verilog [10] et les simulateurs commerciaux associés sont très matures et très efficaces. Mais ils n'effectuent pas le tirage aléatoire des processus propre à la sémantique d'exécution asynchrone.

On peut aussi se tourner vers des langages de description formelle comme SDL [46] dont la sémantique est basée sur les automates communicants ou LOTOS [45] dont la sémantique est basée sur l'approche observationnelle de CCS. Ils fournissent tous deux une composition

synchrone aussi bien qu'une composition asynchrone des processus. Cependant, ces langages bien que standardisés sont beaucoup moins répandus que SystemC v2.0.1 dans le monde de la conception de systèmes sur une puce et de facto beaucoup moins accessibles.

Il y a également une limitation d'ordre théorique. Les structures de contrôle (composition, communication) de ces langages sont concrètes. Ils possèdent donc un pouvoir d'expression plus faible qu'un langage dans lequel ces structures sont abstraites.

SystemC v2.0.1 repose sur un mécanisme ouvert et extensible d'interface pour abstraire les communications entre processus.

Par ailleurs, confirmant cette direction de recherche, deux équipes de chercheurs de l'université technique du Danemark ont collaboré à une étude sur l'adéquation de SystemC v2.0.1 à la modélisation de circuits numériques asynchrones à différents niveaux d'abstraction [47]. Ils se sont concentrés sur l'automatisation du raffinement à partir d'un canal de communication abstrait type CSP vers un canal concret de type deux phases ou quatre phases (voir Figure 9) ; ceci est intéressant sur le plan méthodologique. Mais leur étude est incomplète. Premièrement, ils se focalisent d'emblée sur un type particulier de circuit asynchrone (les circuits micro pipeline) sans vraiment se pencher sur les autres types de circuits (voir paragraphe II5). Deuxièmement, ils ne prennent pas en compte le problème de la vérification, problème crucial dans la conception de circuits numériques. Enfin, ils prétendent à tort que la modélisation de la fourche ou de la synchronisation au sens des réseaux de Petri ordinaires (voir paragraphe I4.2) est impossible.

La question de la modélisation, de la simulation et de la vérification de circuits numériques asynchrones au niveau des portes autant qu'à des niveaux d'abstraction plus élevés dans un environnement de conception de systèmes numériques standard comme SystemC v2.0.1 se pose et est ouverte.

Le reste du chapitre sera consacré à une présentation aussi synthétique que possible du standard SystemC v2.0.1. On insistera tout particulièrement sur le noyau de simulation et sur la structuration à base de canaux de communication abstraits. Ces points sont fondamentaux pour la mise en œuvre de la modélisation, la simulation et la vérification de circuits numériques asynchrones dans le standard SystemC v2.0.1.

2 Open SystemC Initiative

OSCI est l'acronyme de « Open SystemC Initiative » en anglais. Il s'agit d'une association indépendante à but non lucratif regroupant de nombreux acteurs du monde de l'industrie, de l'enseignement et de la recherche ainsi que des individus. Sa mission est de promouvoir SystemC v2.0.1 comme un standard pour la conception de systèmes sur une puce [48].

Ce standard se veut ouvert. Tout un chacun peut devenir membre de l'OSCI gratuitement et accéder à une large source d'informations : code source, documentation, exemples, publications scientifiques et techniques, forum de discussion, …

A l'heure actuelle, SystemC v2.0.1 se compose de trois parties : le cœur du langage [49][50][51], une bibliothèque liée à la méthodologie de conception des systèmes sur une puce [52] et une bibliothèque de vérification [53]. Le cœur du langage s'inspire fortement du langage SpecC développé par le professeur Daniel Gajski et son équipe à l'université de Californie, Irvine [56]. La bibliothèque maître/esclave supporte une méthodologie de conception développée par l'OSCI. La bibliothèque de vérification s'inspire largement de l'environnement de vérification TestBuilder [54] développé par la société Cadence autour de son langage Verilog.

Les possibilités de modélisation de SystemC v2.0.1 au niveau du logiciel sont encore limitées [55] et seront véritablement adressées dans la version 3.0 dont la livraison était programmée pour la fin de l'année 2003. Cette version devrait permettre :

- de créer dynamiquement des processus et des canaux élémentaires,
- de contrôler les processus en externe,
- de modéliser l'ordonnancement des processus,
- de gérer la préemption des processus,
- d'annoter en délai l'activité des processus.

Un groupe de travail a été créé en 2002 pour étudier la modélisation de systèmes analogiques et mixtes s'appuyant sur SystemC v2.0.1 [57][58]. Des prototypes et des exemples sont déjà disponibles, mais la première version de production est prévue pour 2004.

Ainsi, SystemC v2.0.1 est en passe de devenir le premier environnement de conception (modélisation à différents niveaux d'abstraction, simulation et vérification) ouvert et extensible de systèmes sur une puce adressant la modélisation de composants logiciels et matériels numériques, analogiques et mixtes.

3 Architecture

3.1 Présentation générale

L'architecture de l'environnement de conception SystemC v2.0.1 est organisée en couches comme représentées sur la Figure 14. Elles sont présentées successivement du niveau le plus bas vers le niveau le plus haut dans les sections suivantes.

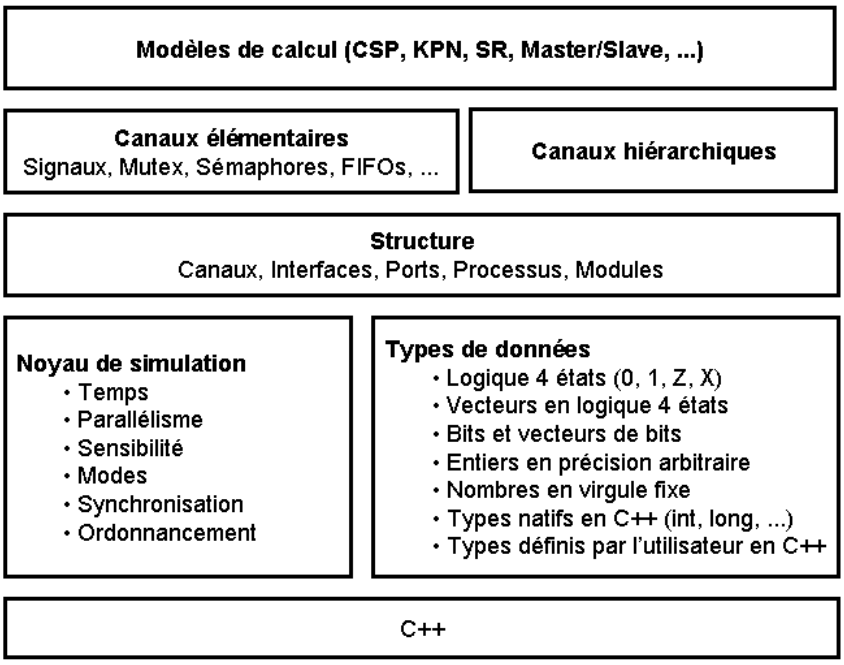


Figure 14 : Organisation en couches de l'architecture de SystemC v2.0.1

La présentation des modèles de calcul cités sur le diagramme est reportée à l'annexe B.

3.2 C++

La couche de plus bas niveau représente la technologie de base choisie pour l'implémentation de SystemC v2.0.1. Il s'agit de C++. Ce langage est standardisé depuis 1998 conformément à la norme ISO/IEC 14882:1998 [59]. Le langage C++ peut-être vu comme une version améliorée du langage C. Les principaux paradigmes de programmation supportés par C++ en font sans doute le langage de programmation le plus évolué à l'heure actuelle :

- programmation procédurale,
- programmation modulaire,
- abstraction des données,
- programmation orientée objet,
- programmation générique.

3.3 Types de données

Se basant sur les trois paradigmes de programmation précédents : abstraction des données, généricité et programmation orientée objet, SystemC v2.0.1 offre des types liés à la programmation aussi bien que des types liés à la description de matériel. Les types liés à la programmation sont les types natifs (booléens, entiers, réels) et les types définis par l'utilisateur (structures et classes). Les types propres aux langages de description de matériel sont les types en logique quatre états assortis de leurs fonctions de résolution pour la représentation de signaux, les types en représentation à virgule fixe pour les applications de traitement du signal et les types entiers en précision arbitraire pour des applications où le codage des nombres dépasse la représentation machine.

3.4 Noyau de simulation

3.4.1 Modèle de temps

SystemC v2.0.1 repose sur un modèle de temps absolu basé sur des mots binaires de 64 bits. La résolution temporelle peut être réglée par un facteur d'échelle et toute valeur inférieure à la résolution est arrondie au plus près. Au cours d'une simulation temporelle, le temps démarre à zéro et ne peut que croître linéairement.

Ce modèle supporte le mode d'exécution temporisé synchrone ainsi que le mode d'exécution asynchrone à la résolution temporelle près qui fixe la distance minimale entre deux événements. Le simulateur supporte aussi une relation d'ordre totale ou une relation d'ordre partielle propres aux simulations non temporisées à travers les notions d'événement immédiat et d'événement à temps nul qui seront présentées au paragraphe 3.4.6.

3.4.2 Parallélisme

Le noyau de simulation de SystemC v2.0.1 est basé sur le mécanisme des co-routines [61].

Ce mode d'exécution repose sur le principe de séparation des programmes. On dit d'un programme qu'il est séparable si on peut le découper en unités de traitement ou modules qui communiquent les uns avec les autres pourvu que :

- l'information échangée entre les modules soit de nature discrète,
- l'information entre modules circule dans des canaux unidirectionnels,

- le programme tout entier puisse être disposé de telle sorte que les entrées se retrouvent à gauche et les sorties à droite et partout ailleurs l'information circule de la gauche vers la droite.

Dans ces conditions, chaque module peut être implémenté comme une co-routine, i.e. un programme autonome qui communique avec ses voisins comme s'ils étaient des sous-routines d'entrée ou de sortie. Ainsi, les co-routines sont toutes des sous-routines au même niveau de hiérarchie, chacune se comportant comme le programme principal. A un instant donné plusieurs modules peuvent être candidats à l'exécution simultanément. C'est le rôle du mécanisme d'ordonnancement du simulateur de gérer cela ; ce qu'on détaillera au paragraphe 3.4.7. Il n'y a pas de restriction a priori quant au nombre d'entrées/sorties d'une co-routine.

La notion de co-routine se prête particulièrement bien à la conception de processus communicant de manière asynchrone. En effet, comme on le voit sur la Figure 15, deux co-routines A et B sont interconnectées de sorte que A envoie des informations à B. Supposons par exemple que B s'exécute en premier jusqu'à rencontrer une commande de lecture. B stoppe son exécution sur la primitive « wait() », rend la main au simulateur et attend une information de la part de A. Ensuite dans cet exemple le contrôle ne peut être transféré qu'à A qui s'exécute jusqu'à rencontrer une commande d'écriture. Ensuite A se bloque et le contrôle est transféré de nouveau à B qui peut lire l'information présente sur le canal. Le tout fonctionne sans horloge globale.

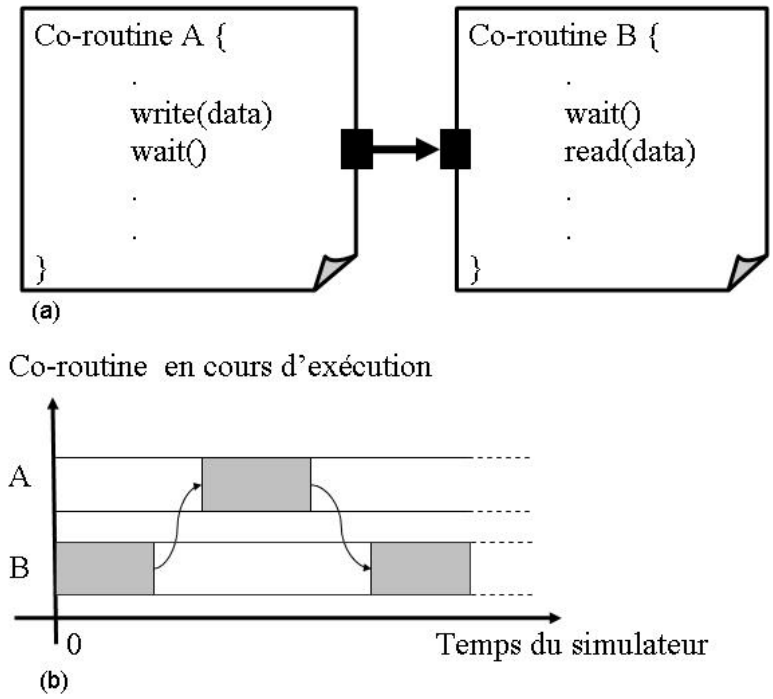


Figure 15 : (a) Interconnexion de co-routines. (b) Diagramme d'exécution

3.4.3 Initialisation

SystemC v2.0.1 fournit un mode de contrôle de l'initialisation des co-routines pendant la phase d'élaboration par l'intermédiaire de la fonction « dont_initialise() ». Si cette fonction est utilisée au moment de la déclaration d'une co-routine pendant la phase d'élaboration, cette

dernière est configurée de manière à ne pas être activée lors de la phase d'initialisation du simulateur.

3.4.4 Sensibilisation des processus

SystemC v2.0.1 propose deux modes de sensibilisation des processus sur les événements : un mode statique et un mode dynamique. Le mode statique fabrique le lien entre les événements et les processus avant le démarrage de la simulation. Le mode dynamique fabrique le lien entre les événements et les processus pendant la simulation sur les points de synchronisation des processus avec le noyau de simulation, reconfigurant éventuellement les liens définis soit statiquement pendant la phase d'élaboration, soit dynamiquement dans une étape précédente de la simulation.

3.4.5 Modes d'exécution des co-routines

Le noyau de simulation de SystemC v2.0.1 fournit deux modes d'exécution des co-routines.

Le premier mode est fonctionnel. Les co-routines « SC_METHOD » sont modélisées par des fonctions C++. Leur exécution correspond à un appel de fonction. A l'initialisation, une co-routine fonctionnelle est systématiquement exécutée indépendamment des points de synchronisation avec le noyau. Le transfert du contrôle de l'exécution à une autre co-routine se fait systématiquement à la fin de l'exécution de la fonction. La sensibilité peut être modifiée par un appel à la primitive « next_trigger() » qui est évaluée à la fin de l'exécution de la fonction. Si plusieurs appels à cette primitive sont effectués en séquence, seul le dernier appel est conservé. Conformément à la sémantique d'exécution des fonctions en C++, les variables locales utilisées pendant l'exécution sont perdues à la fin du calcul.

Le second mode est contextuel. Un contexte d'exécution est créé pour chaque co-routine « SC_THREAD » à l'initialisation un peu comme pour un processus dans un système d'exploitation. Ce contexte réside en mémoire tout au long de la simulation. A l'initialiation, une co-routine contextuelle est exécutée jusqu'au premier appel à la primitive de synchronisation avec le noyau de simulation. Le transfert du contrôle de l'exécution à une autre co-routine peut se faire n'importe où dans le corps de la co-routine par un appel à la primitive de synchronisation « wait() ». Les variables locales utilisées pour le calcul pendant l'exécution de la co-routine sont conservées par le contexte.

Dans les deux cas, la liste de sensibilité des co-routines est soit statique, soit dynamique. Pourtant, le mécanisme de gestion de la liste de sensibilité des co-routines dans un langage de description de matériel est en général statique. Car les structures matérielles sont le plus souvent figées; ceci fait que les modules intégrés sont sensibles à une liste prédéterminée de signaux au moment de l'élaboration. Ce n'est plus le cas pour un bloc de logique programmable ou une pile de processus dans un système d'exploitation. Les deux mécanismes peuvent être utilisés de concert.

Aussi bien dans un mode que dans l'autre, il est impossible pour le simulateur d'interrompre l'exécution d'une co-routine une fois que celle-ci est déclenchée. Le noyau de simulation de SystemC v2.0.1 est non préemptif.

La Figure 16 résume ces différentes caractéristiques.

Mode	Initialisation	Sensibilisation	Sauvegarde du contexte	Transfert du Contrôle	Préemption
Fonctionnel	Exécution complète de la fonction	Statique : « sensitive() » Dynamique : « next_trigger() »	Non	Fin d'exécution de la fonction	Non
Contextuel	Exécution de la fonction jusqu'au premier appel à « wait() »	Statique : « sensitive() » Dynamique : « wait() »	Oui	sur « wait() » ou à la fin de l'exécution de la fonction	Non

Figure 16 : Modes d'exécution des co-routines par le noyau de simulation de SystemC v2.0.1

On trouvera à l'annexe B.3 des exemples de code SystemC v2.0.1 déclinant ces concepts sur une porte non-et (« nand » en anglais).

3.4.6 Synchronisation

3.4.6.1 Introduction

La notion de synchronisation est intimement liée au concept d'événement et de communication. Dans un système à événements discrets, les événements sont atomiques, distinguables et de durée nulle. SystemC v2.0.1 les considère unidirectionnels contrairement à d'autres cadres dans lesquels ils sont considérés comme bidirectionnels (cf. CSP [39], LOTOS [45]).

3.4.6.2 Notification d'occurrences d'événements

Dans SystemC v2.0.1, un événement est modélisé par la classe C++ « sc_event » et une occurrence d'événement peut être notifiée de trois façons différentes depuis une co-routine :

- immédiatement,
- après un délai nul,
- après un délai non nul.

Il se peut qu'au cours d'une simulation, plusieurs occurrences du même événement soient notifiées depuis des co-routines différentes. Dans ce cas, une notification immédiate a priorité sur une notification à délai nul ayant elle-même priorité sur une notification à délai non nul. Et seule la notification de priorité la plus haute est conservée.

La syntaxe de ces notifications est rappelée à l'annexe B.4.

3.4.6.3 Déclenchement des co-routines sur occurrences d'événements

Les co-routines peuvent être déclenchées :

- sur l'occurrence d'un événement,
- sur la conjonction de l'occurrence de plusieurs événements,
- sur la disjonction de l'occurrence de plusieurs événements,
- après un délai,
- sur l'occurrence d'un événement ou après un délai donné,
- sur la conjonction de l'occurrence de plusieurs événements ou après un délai donné,
- sur la disjonction de l'occurrence de plusieurs événements ou après un délai donné.

Les trois derniers modes correspondent au mode minuterie (« timer » en anglais).

La même syntaxe est valable pour la primitive « next_trigger() » dans le cas des co-routines fonctionnelles.

On pourra se reporter à l'annexe B.5 pour les détails de la syntaxe de la primitive de synchronisation des processus sur occurrences d'événements.

3.4.7 Ordonnancement

L'algorithme d'ordonnancement implémenté dans le noyau de simulation de SystemC v2.0.1 est relativement rudimentaire (voir Figure 17). Il n'y a pas de préemption, ni de notion de priorité entre les co-routines. C'est plutôt le rôle d'un système d'exploitation que de gérer ce type de propriétés et c'est prévu pour les versions futures de SystemC v2.0.1, notamment la version 3.0 focalisée sur la modélisation de logiciel [55].

Cet algorithme fonctionne à partir d'une pile de co-routines, d'une pile d'occurrences d'événements et d'une pile de variables à mettre à jour pour les canaux élémentaires. Il démarre à la fin de la phase d'élaboration. A l'initialisation, la pile de co-routines est chargée avec les co-routines à initialiser. Toutes les co-routines prêtes à être exécutées sont évaluées dans un ordre indéterminé, puis placées en attente. Ensuite, la mise à jour des variables (cela ne concerne que les canaux élémentaires) a lieu. S'il reste des notifications immédiates dans la pile d'événements, les co-routines correspondantes sont déclenchées et l'évaluation des co-routines reprend jusqu'à ce qu'il n'y ait plus de co-routine dans la pile de co-routine. Ensuite, s'il reste des notifications à temps nul dans la pile d'événements, les co-routines correspondantes sont déclenchées et l'évaluation des co-routines reprend jusqu'à ce qu'il n'y ait plus d'occurrence d'événement à temps nul dans la pile. Puis, le temps progresse jusqu'à l'événement à délai non nul le plus proche. Les co-routines correspondantes sont déclenchées et l'évaluation des co-routines reprend jusqu'à ce que le temps atteigne la borne supérieure spécifiée au lancement de la simulation.

On peut noter que les commandes d'élaboration et de simulation ne sont pas séparées. Par contre, un mécanisme de rappel est prévu (fonction « end_of_elaboration() ») et permet d'intervenir une fois la phase d'élaboration par défaut effectuée. Cette phase d'élaboration par défaut est définie dans le manuel de référence du langage [49] comme l'exécution du programme principal depuis son appel jusqu'au premier appel de la fonction « sc_start » chargée de démarrer la simulation.

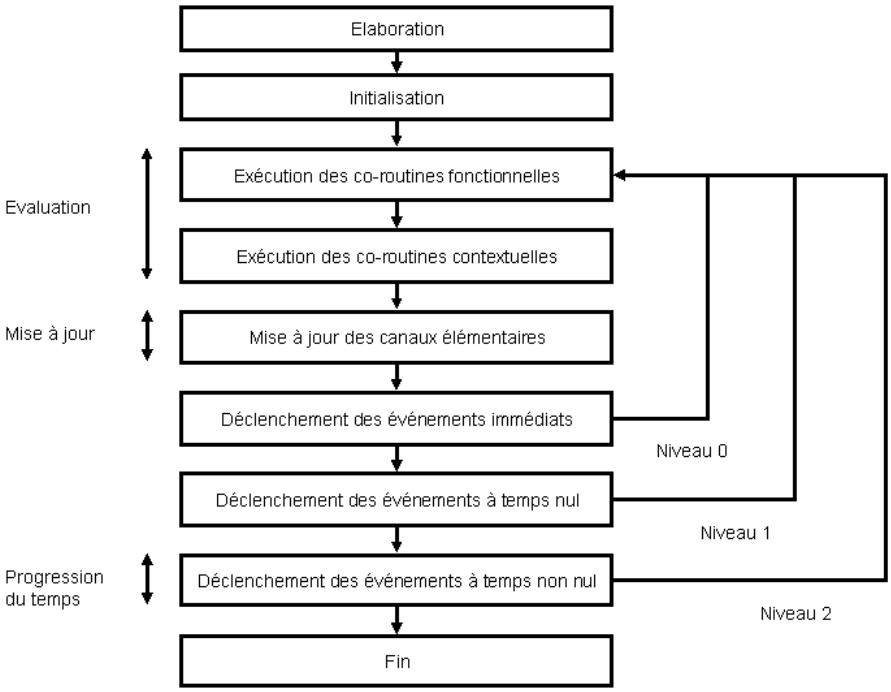


Figure 17 : Diagramme de fonctionnement du noyau de simulation de SystemC v2.0.1

Sur le diagramme, on a pris soin de numéroter les boucles d'évaluation des co-routines de la plus profonde à la moins profonde. La boucle la plus profonde correspond aux notifications immédiates et porte le nom de niveau 0. La boucle intermédiaire correspond aux notifications à délai nul et porte le nom de niveau 1. La boucle la moins profonde correspond aux notifications à délai non nul et porte le nom de niveau 2. On utilisera cette numérotation tout au long du document.

3.5 Structure

3.5.1 Eléments de base

Conformément à la plupart des langages de description de matériel, SystemC v2.0.1 propose de modulariser et de hiérarchiser la description d'un système.

Les éléments de base de la description sont les modules et les canaux de communication.

Les modules encapsulent la déclaration de ports, de processus, de variables partagées entre les processus et d'instances d'autres modules interconnectés par des canaux, supportant ainsi la hiérarchisation de la description du système.

Les canaux de communication implémentent des interfaces abstraites de communication. Ils encapsulent les méthodes permettant d'assurer la synchronisation et le partage des variables entre les processus encapsulés dans les modules. Ces canaux sont de deux types : soit élémentaires, soit hiérarchiques.

Pour saisir ce concept de communication abstraite, on peut faire une analogie entre données et communication. Dans un langage où les données sont abstraites, on a des types de données natifs et des types de données définis par l'utilisateur.

Par exemple, en C/C++ on a des entiers (mot-clé « int »). C'est un type natif codé sur 32 bits dont la représentation machine est très efficace. Toujours en C/C++, on a des structures (mot-clé « struct ») qui permettent au programmeur de composer des types complexes.

Dans un langage où la communication est abstraite comme en SpecC ou SystemC v2.0.1, on a des canaux élémentaires et des canaux hiérarchiques. Les canaux élémentaires correspondraient aux types de données natifs alors que les canaux hiérarchiques correspondraient aux types de données définis par l'utilisateur.

3.5.2 Composition des éléments de base

La Figure 18 montre une représentation abstraite d'un circuit dans SystemC v2.0.1. Ce circuit est composé d'un module émetteur, d'un module récepteur et d'un canal de communication.

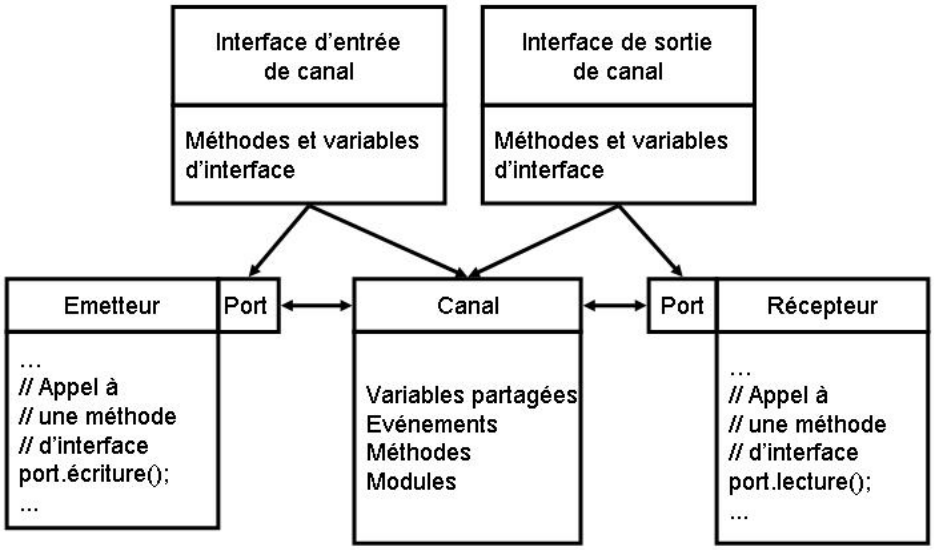


Figure 18 : Représentation abstraite d'un canal de communication dans SystemC v2.0.1

Les processus dans les modules accèdent aux canaux à travers les ports des modules. Le programme principal, lui, accueille la définition de la résolution temporelle, la déclaration du sommet de la hiérarchie des modules, la définition du fichier de traces et la fonction de lancement de la simulation.

Le flot actuel de conception dans SystemC v2.0.1 permet de modéliser et de simuler un système à événements discrets fermé embarquant dans un programme compilé la machine, les stimuli, les sondes et le simulateur comme sur la Figure 4.

On développe à l'annexe B.2 la modélisation SystemC v2.0.1 d'un oscillateur en anneau basé sur le canal élémentaire « sc_signal ». Ce canal représente le signal matériel dans SystemC v2.0.1.

IV Modélisation et simulation de circuits numériques asynchrones dans SystemC v2.0.1

L'objectif de ce chapitre de contribution est multiple. Nous adaptons d'abord la sémantique de SystemC v2.0.1 aux circuits numériques asynchrones, après avoir soigneusement étudié les limitations du standard. Puis, nous établissons une structure type de circuit dans SystemC v2.0.1 basée sur le paradigme original de programmation de SystemC v2.0.1 permettant de séparer calcul et communication. Ensuite, nous focalisons sur les circuits quasi insensibles aux délais que nous formalisons en LTL. Nous appliquons ce principe de modélisation basée sur une structure type de circuit dans SystemC v2.0.1 aux éléments des circuits numériques asynchrones formalisés en LTL. Nous complétons le modèle obtenu par les éléments producteur et consommateur chargés de l'initialisation de la simulation et de la modélisation de l'environnement du circuit. Il faut effectivement garder à l'esprit l'absence d'horloge globale cadençant l'évolution du circuit. Nous étudions la temporisation du modèle. Cette temporisation doit être cohérente avec le comportement du système avant et après prise en compte des délais. Enfin, nous mettons en oeuvre le modèle obtenu sur un circuit de taille réduite, mais significatif. La contribution de ce chapitre est un modèle de circuits numériques asynchrones, s'intégrant dans le standard SystemC v2.0.1 de conception de systèmes numériques.

1 Introduction

1.1 Limitations des standards

Les langages standard de spécification (SDL, LOTOS) et les langages standard de description de matériel (VHDL, Verilog) possèdent des structures concrètes de communication : le signal en VHDL, les opérateurs de composition en LOTOS. Ces structures syntaxiques sont figées dans le langage.

Par contre, dans SystemC v2.0.1, les interfaces de communication sont abstraites. Les canaux de communication implémentent des interfaces abstraites de communication. La communication ne repose que sur deux types d'objets : l'événement et le processus. De là découlent deux caractéristiques très importantes aussi bien pour le programmeur en SystemC v2.0.1 que pour le concepteur de SystemC v2.0.1 lui-même. D'une part, le calcul et la communication sont découplés. On a d'un côté, le traitement de l'information réalisé par les processus et de l'autre, le protocole d'échange de l'information réalisé par les canaux. D'autre part, le modèle de communication choisi par le programmeur en SystemC v2.0.1 est découplé du simulateur lui-même puisque la sémantique de SystemC v2.0.1 considère des communications abstraites.

Héritant de C++ et de SpecC, SystemC v2.0.1 possède alors en tant que langage de programmation deux caractéristiques fondamentales : l'abstraction des données et l'abstraction du contrôle.

Il s'agit du langage standard de description de matériel au pouvoir d'expression le plus fort à l'heure actuelle. Il offre un paradigme de programmation original dans lequel il est possible de découpler traitement et échange de l'information aussi bien entre les éléments d'un modèle qu'entre un modèle et le noyau de simulation.

1.2 Limitations de SystemC v2.0.1

1.2.1 Modèle de temps

SystemC v2.0.1 supporte le modèle de temps des systèmes à événements discrets synchrones. Qu'en est-il des systèmes à événements discrets asynchrones ?

Dans le cas des circuits numériques asynchrones en particulier, deux types de délais sont possibles : des délais non bornés et des délais bornés (voir chapitre II). Or le noyau de simulation de SystemC v2.0.1 repose sur des délais codés en binaire sur 64 bits, donc bornés (voir paragraphe III3.4.1).

Est-ce réellement une limitation ?

Nous répondons à cette question au paragraphe 2.1.

1.2.2 Puissance d'expression

SystemC v2.0.1 propose un paradigme de programmation original dans lequel calcul et communication sont séparés. Que cela veut-il dire exactement ? Comment pouvons-nous tirer profit de ce mécanisme dans le cadre de la modélisation de circuits numériques asynchrones ? La puissance d'expression offerte par ce mécanisme est-elle suffisante ? Une étude détaillée du mécanisme de synchronisation de SystemC v2.0.1 permet de répondre à ces questions. C'est l'objet de ce paragraphe.

SystemC v2.0.1 propose une algèbre de calcul sur les événements. On peut synchroniser un processus sur un événement, sur une disjonction d'événements ou sur une conjonction d'événements.

Comme on vient de l'évoquer, on dispose des opérateurs « ou » et « et » sur les occurrences d'événement. On dispose également de la « fourche » comme nous le voyons par la suite. Par contre, il manque la structure de « choix » pour atteindre la puissance d'expression d'un réseau de Petri ordinaire (voir I4).

De plus, pour que cette algèbre de calcul sur événement soit complète au sens de la machine de Turing, encore faudra-il lui adjoindre l'opérateur de détection de l'absence d'un événement. Cet opérateur correspondrait au « non » logique. Si nous disposions de cet opérateur, nous pourrions réaliser tous les calculs dont on a généralement besoin en informatique simplement en propageant des événements à travers un réseau de processus.

Dans ce cas, calcul et communication seraient totalement confondus. Cette approche correspond cependant à un processus de conception exotique. Dans le language commun, quel sens donner au déclenchement d'une action sur la disparition d'un événement ? Ce n'est pas très naturel.

Il sera préférable pour atteindre la complétude au sens de la machine de Turing de séparer calcul et communication comme on le verra au paragraphe 2.1.

1.2.3 Capacités de modélisation

On a passé en revue au chapitre III les capacités de modélisation du standard SystemC v2.0.1. De nombreux choix sont offerts, tant sur le plan de la modélisation, que sur le plan de la simulation.

Rien qu'au niveau des processus, on peut choisir d'initialiser une co-routine ou non, d'utiliser une co-routine contextuelle ou une co-routine fonctionnelle, de recourir à des variables

partagées au niveau du module, d'utiliser le mécanisme de sensibilisation statique ou le mécanisme de sensibilisation dynamique des co-routines.

Quels sont les tenants et les aboutissants de ces différentes caractéristiques vis-à-vis de notre problématique ?

Nous répondons à cette question au paragraphe 2.3.

1.2.4 Indétermination du comportement

1.2.4.1 Tirage aléatoire des processus

Le manuel de référence du langage SystemC v2.0.1 [49] stipule que l'ordre d'exécution des processus n'est pas spécifié. Par contre, deux simulations utilisant le même simulateur doivent rendre le même résultat.

Cependant, le document de spécification fonctionnelle [50] annonce que l'ordre d'exécution des processus doit pouvoir varier d'une simulation à l'autre. En effet, dans certains modèles de calcul, notamment dans les modèles de calcul asynchrones, il faut prendre en compte le caractère aléatoire du comportement. Dans ce document, il est précisé qu'une option sur la ligne de commande peut être passée au simulateur pour tirer aléatoirement les processus à chaque simulation.

A moins d'intégrer dans la définition de « rendre le même résultat » la caractéristique aléatoire du modèle de calcul considéré, on ne peut que conclure que les deux documents susdits sont contradictoires.

Mais ce n'est pas tout. Lorsque l'on regarde le code de plus près, on constate que l'option de tirage aléatoire des processus n'est pas implémentée dans SystemC v2.0.1.

Du coup, on n'a pas de contrôle sur l'ordre d'exécution des processus. Cet ordre n'est pas spécifié par définition. Il dépend globalement de l'environnement de travail (station de travail, compilateur C++, …).

1.2.4.2 Risque de défaut de modélisation

Par ailleurs, le manuel de référence du langage SystemC v2.0.1 [49] indique clairement un risque de défaut sur le plan de la modélisation. La Figure 19 montre un exemple mettant en évidence ce risque d'indétermination du comportement provenant de la modélisation. Si la co-routine A s'exécute avant la co-routine B, alors la co-routine C est immédiatement activée. Puis la co-routine B est activée et C est activée encore une fois. C est donc activée deux fois. Par contre, si la co-routine B est exécutée d'abord, l'occurrence correspondante est écrasée par l'occurrence de l'événement de la co-routine A à cause de la règle de priorité entre les niveaux de simulation et C n'est plus activée qu'une seule fois.

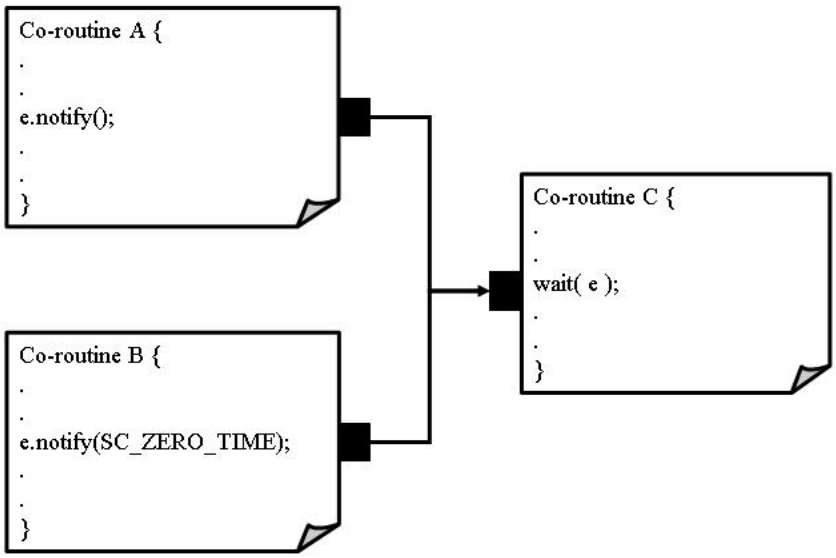


Figure 19 : Risque d'indétermination du comportement à partir de la modélisation

Quelles conclusions peut-on tirer de cette analyse ?

Nous répondons à ces questions au paragraphe 2.4.

1.3 Choix d'un type de circuit numérique asynchrone

Nous proposons d'organiser notre réflexion autour des circuits quasi insensibles aux délais. Comme nous l'avons évoqué au paragraphe II3, ce modèle a la puissance d'expression de la machine de Turing. De plus, nous proposons une formalisation de ses éléments (garde, règle de production et portes) ainsi que des propriétés d'insensibilité aux délais qu'il doit vérifier (persistance, sûreté, vivacité) dans le cadre de LTL. Ce modèle s'inscrit donc dans un cadre rigoureux et complet au sens de la machine de Turing.

1.4 Contribution

Forts de ces analyses, nous profitons du caractère ouvert du standard SystemC v2.0.1 pour modifier le noyau de simulation en y introduisant d'une part un drapeau indiquant si l'occurrence d'un événement a eu lieu ou pas et d'autre part le tirage aléatoire des processus qu'impose la sémantique des systèmes à événements discrets asynchrones.

Ce faisant, nous identifions un point délicat, l'impact du tirage aléatoire des processus sur le mécanisme d'élaboration du simulateur. Nous en dérivons des contraintes de modélisation et de simulation qui permettent de supporter exactement la sémantique des systèmes à événements discrets asynchrones dans le standard SystemC v2.0.1.

Nous formalisons en logique linéaire temporelle les circuits quasi sensibles aux délais comme point de départ rigoureux et complet de notre réflexion sur les circuits numériques asynchrones.

Nous tirons parti du paradigme de programmation de SystemC v2.0.1 (séparation du calcul et de la communication) présenté au paragraphe III3.5 pour modéliser les éléments d'un circuit quasi insensible aux délais en se basant d'une part sur des modules pour les règles de production et les portes et d'autre part sur un canal de communication hiérarchique point à

point unidirectionnel pour la synchronisation et l'échange de données entre ces éléments. Ce canal supporte deux modes de fonctionnement : le mode direct et le mode à poignée de main. Ce faisant, nous identifions un point délicat : le problème de l'initialisation de la simulation. Cela nous amène à compléter le modèle des circuits quasi insensibles aux délais avec deux éléments : le producteur et le consommateur.

Dans un but didactique, nous présentons dans un premier temps un modèle non temporisé de circuits dans SystemC v2.0.1. Cela pose les bases de notre modèle de circuits numériques asynchrones. Dans un second temps, nous étudions la temporisation du modèle.

Enfin, nous illustrons ces concepts au paragraphe 8 à travers un exemple de circuit numérique asynchrone insensible aux délais de taille réduite, mais significatif : l'oscillateur de Muller [11].

Notre flot de conception concerne les 4 phases suivantes du flot de conception descendant de circuits numériques :

- modélisation non temporisée,
- simulation non temporisée,
- temporisation du modèle,
- simulation temporisée.

Deux difficultés apparaîtront clairement. Dans le cas d'une simulation de niveau 0, on verra apparaître la rupture de la relation de causalité à l'initialisation. Dans le cas d'une simulation temporisée, on verra que le dispositif de mise à zéro doit être temporisé avec soin. A chaque fois, il s'agit de vérifier l'insensibilité aux délais du modèle.

Nous serons mieux à même de cerner ces difficultés au chapitre V, une fois que nous aurons instrumenté notre modèle de circuits numériques asynchrones pour vérifier les propriétés d'insensibilité aux délais, i.e. persistance, sûreté et vivacité.

2 Adaptation de la sémantique de SystemC v2.0.1

2.1 Modèle de temps

On pourrait utiliser un type de données indépendant de la représentation machine pour la représentation du temps. On pourrait ainsi ajouter de nouveaux pas de calcul au cours de la simulation au besoin. Ainsi, le modèle de temps serait non borné (aux capacités de calcul de la station de travail près). Cela se ferait au détriment des performances du simulateur. Nous ne choisissons pas cette approche.

Dans le cas d'une simulation non temporisée, on dispose de plus de 16 milliards de milliards de pas de calcul. Dans le cas d'une simulation temporisée, on dispose de plus de 5 heures simulées à l'échelle de la femto seconde.

Nous considérons par conséquent que la résolution et la dynamique du modèle de temps de SystemC v2.0.1 sont suffisantes.

2.2 Puissance d'expression

2.2.1 Production-consommation d'occurrences d'événements

Comme nous l'avons évoqué au paragraphe 1.2.2, une structure de « choix » au sens des réseaux de Petri est sensible à une conjonction d'événements. Un calcul particulier est déclenché en fonction de l'événement reçu. Cela implique que l'on soit capable d'identifier le ou les événements responsables du déclenchement d'un processus pour exécuter la branche du choix correspondante. Or, cette information sur l'origine de l'événement ou des événements responsables du déclenchement du processus n'est pas disponible dans la version actuelle de SystemC v2.0.1.

Pour compenser cette limitation, on introduit dans la classe « événement » du noyau de simulation de SystemC v2.0.1 un drapeau indiquant si l'occurrence d'un événement a eu lieu ou pas. On dira que si le drapeau est levé, une occurrence d'événement est produite. On dira que si le drapeau est baissé, une occurrence d'événement est consommée. A l'initialisation, tous les drapeaux sont baissés. Cela conduit à l'algorithme de simulation représenté sur la Figure 20.

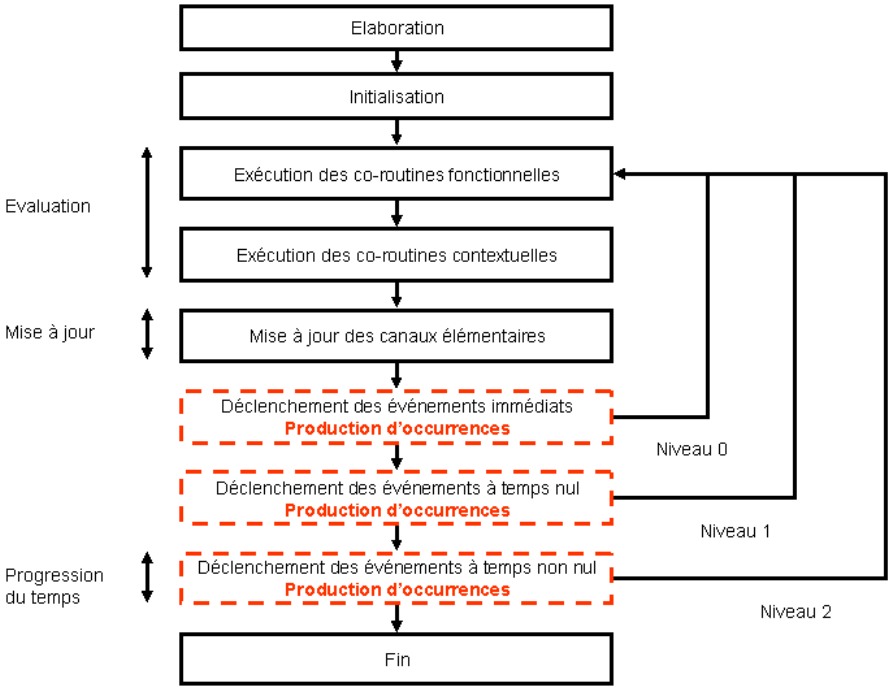


Figure 20 : Production d'occurrences d'événements dans le noyau de simulation de SystemC v2.0.1

Ce n'est que lorsque que le noyau de simulation lève le drapeau à la suite d'une occurrence d'événement que l'on dit qu'il y a une production d'occurrence d'événement.

On réinitialise ce drapeau depuis le modèle une fois une telle occurrence détectée. Cela conduit à un protocole de communication production-consommation entre le modèle et le noyau de simulation comme sur la Figure 21.

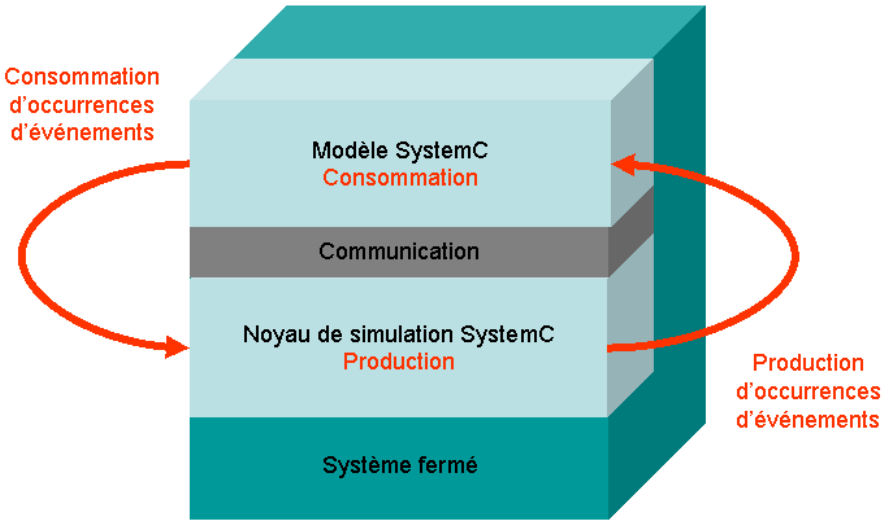


Figure 21 : Production-consommation d'occurrences d'événements entre un modèle et le noyau de simulation SystemC v2.0.1

Le pseudo code de l'annexe C.1 illustre ce nouveau comportement, en faisant abstraction des aspects structurels (module, interface, canal).

2.2.2 Séparation du calcul et de la communication

Il nous reste encore à atteindre la puissance d'expression de la machine de Turing. Plutôt que d'introduire l'opérateur de détection de l'absence d'un événement, comme on l'a évoqué au paragraphe 1.2.2, on préfère découpler calcul et communication. Cela évite d'alourdir l'algorithme de simulation. Il devient plus commode d'élever le niveau d'abstraction du modèle. Cela correspond à processus de conception plus naturel.

On dispose de modules pour le calcul et de canaux pour la communication. Un canal contient des variables partagées entre les modules. Parmi ces variables, on distingue une valeur et un événement. La valeur représente une information numérique échangée entre les modules. L'événement représente un signal de synchronisation entre les modules. Un événement est émis à chaque fois que l'information transférée change. Le drapeau d'occurrence associé à un événement permet de repérer depuis un module si une information a changé ou pas.

Par exemple, on associe le type booléen à la valeur véhiculée par le canal et on décide que la valeur « vrai » représente la présence d'un événement et la valeur « faux » l'absence d'un événement. C++ fournit l'opérateur « non » logique sur une valeur booléenne. Nous voilà donc dotés des opérateurs « ou », « et » et « non » sur les événements et nous disposons alors du minimum minimorum pour modéliser les circuits quasi insensibles aux délais tout en bénéficiant de la séparation du calcul et de la communication.

Un exemple de pseudo code de canal de communication point à point unidirectionnel mettant en évidence cette séparation du calcul et de la communication ainsi que son utilisation dans un module en prenant en compte le protocole de production-consommation des événements entre le modèle et le noyau de simulation SystemC v2.0.1 est présenté à l'annexe C.2.

2.3 Capacités de modélisation

Du point de vue du matériel, dès la mise sous tension, n'importe quel composant est susceptible de produire une action, qu'elle dérive d'un comportement spécifié ou non, correct ou non.

Nous nous plaçons par conséquent dans le cas où toutes les co-routines sont candidates à l'exécution dès la phase d'initialisation.

Les comportements des co-routines dans le mode fonctionnel ou bien dans le mode contextuel sont équivalents. On dispose en effet de variables partagées entre les co-routines au niveau d'un même module. Cela permet de reproduire le comportement du mode contextuel à l'aide du mode fonctionnel. Par conséquent, on peut faire abstraction du mode d'exécution des co-routines.

Dans la suite, on ne parle donc plus de co-routine, mais de processus.

La question de la sensibilisation des processus est plus délicate et exige un développement plus long. Nous lui réservons le paragraphe 2.5.

2.4 Indétermination du comportement

2.4.1 Risque de défaut de modélisation

Pour éliminer le risque de défaut dans la modélisation présenté au paragraphe 1.2.4, il suffit d'interdire l'émission d'occurrences immédiates et d'occurrences à délai nul du même événement à partir de processus en parallèle communicant dans la même direction sur le même canal. C'est une contrainte de modélisation que nous reprenons au paragraphe 2.6.

2.4.2 Tirage aléatoire des processus

Pour modéliser correctement la sémantique d'exécution des systèmes à événements discrets asynchrones (voir paragraphe I4.3), le noyau de simulation doit tirer aléatoirement un processus parmi tous ceux candidats à l'exécution à un moment donné.

Nous implantons donc un mécanisme de tirage aléatoire des processus dans le noyau de simulation (voir Figure 22). Ce mécanisme repose sur un générateur de nombres aléatoires linéaire à congruence sur 48 bits initialisé par défaut avec la date courante au début de la simulation.

Avant chaque évaluation de la pile des processus, on génère une permutation ensuite utilisée pour récupérer l'index du processus à exécuter. Sur le plan pratique, pour ne pas trop affecter les performances, on choisit de générer cette permutation en tête de la phase d'évaluation. Le générateur est réinitialisé après un nombre donné de tirages avec le dernier nombre généré par défaut. Il faut en effet tenir compte de la période du générateur pseudo aléatoire.

Un exemple de pseudo code du programme principal avec activation du tirage aléatoire des processus pour le programme principal est donné à l'annexe B.6.

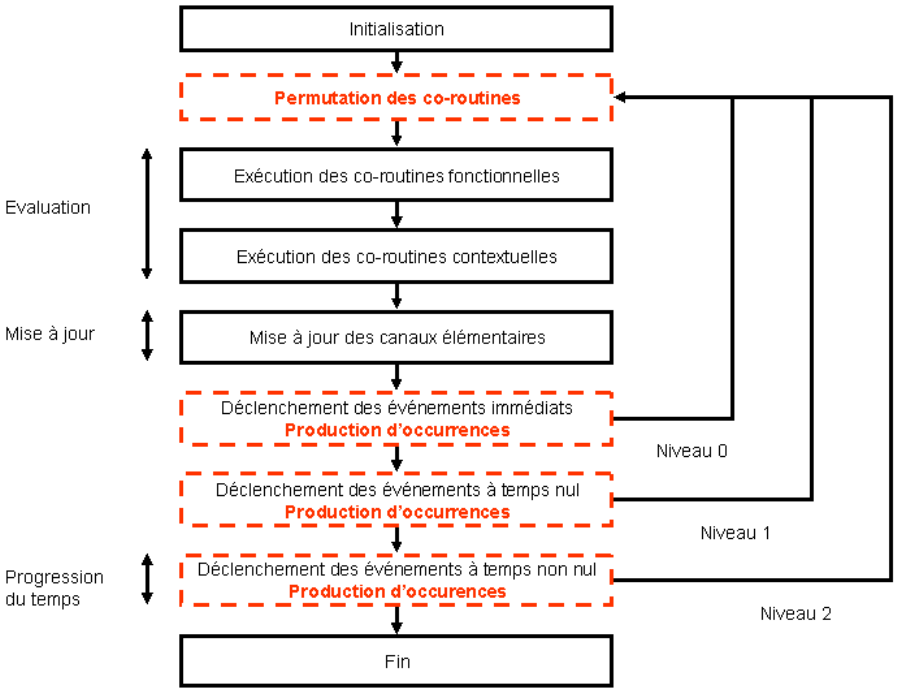


Figure 22 : Introduction du tirage aléatoire des processus dans le noyau de simulation de SystemC v2.0.1

2.5 Sensibilisation des processus

2.5.1 Introduction

Comme on l'a évoqué au paragraphe 2.3, le choix d'un mode de sensibilisation des processus est un problème délicat. On a le choix de connecter les événements aux processus soit au moment de l'élaboration, soit lors de la simulation, soit de composer les deux mécanismes. Quelle relation y a-t-il entre tirage aléatoire, sensibilisation et modélisation des processus sachant que ceux-ci sont tous candidats à l'exécution à l'initialisation ?

Pour répondre à cette question, regardons ce qu'il se passe lorsque le tirage aléatoire des processus est activé. Considérons la spécification CSP décrite par l'équation $P = A \rightarrow B$. Le système P exécute l'action A, puis l'action B, puis s'arrête. Nous proposons de modéliser le comportement de ce modèle par deux modules SystemC v2.0.1 hébergeant chacun un processus. Ces modules sont reliés par le canal de communication unidirectionnel point à point C, comme le montre la Figure 23.

La seule contrainte de modélisation sur l'action A consiste à émettre une requête. La seule contrainte de modélisation sur l'action B consiste à se synchroniser sur la requête provenant de A. Dans le cas général, on considère que le processus A effectue un précalcul avant l'émission de la requête et un postcalcul après l'émission de la requête. De même, on considère que le processus B fait un précalcul avant la synchronisation sur la requête et un postcalcul après la synchronisation sur la requête.

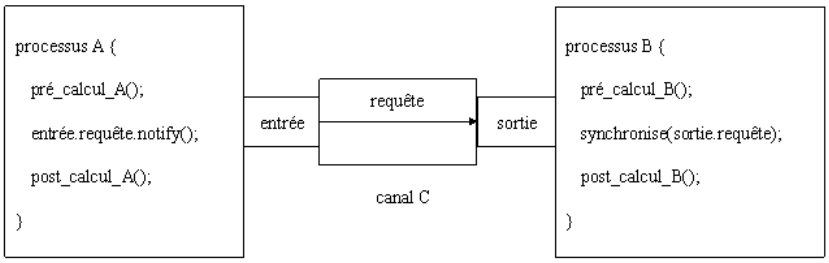


Figure 23 : Modèle SystemC v2.0.1 de la spécification $P = A \rightarrow B$

2.5.2 Sensibilisation statique des processus

Considérons dans un premier temps une sensibilisation statique des processus. Pendant la phase d'élaboration, les liens entre les processus et les événements ainsi que les liens entre les événements et les processus sont créés. A l'initialisation, tous les processus, i.e. A et B sont candidats à l'exécution.

Puisque le tirage aléatoire des processus est activé, on peut obtenir plusieurs exécutions suivant que l'on commence la simulation par l'action A ou par l'action B.

Si on commence par A, on exécute A, qui déclenche B, puis B. Dans ce cas, il y a coïncidence entre le déclenchement de B du fait de la requête et le déclenchement de B du fait de l'initialisation. Cela conduit au comportement suivant :

```
1 pré_calcul_A() ;
2 notification(requête) ;
3 post_calcul_A() ;
4 pré_calcul_B() ;
5 synchronise(requête) ;
6 post_calcul_B() ;
```

Si on commence par B, on exécute B à l'initialisation, puis A qui déclenche B, puis B. Cela conduit au comportement suivant :

```
1 pré_calcul_B() ;
2 synchronise(requête) ;
3 pré_calcul_A() ;
4 notification(requête) ;
5 post_calcul_A() ;
6 pré_calcul_B() ;
7 synchronise(requête) ;
8 post_calcul_B() ;
```

La spécification n'est pas respectée. En effet, nous n'avons spécifié qu'un seul comportement et on en obtient deux. Le premier comportement correspond exactement à la spécification. Le second comportement ne répond pas à la spécification.

2.5.3 Sensibilisation dynamique des processus

Considérons dans un second temps la sensibilisation dynamique des processus. Pendant la phase d'élaboration, seuls les liens entre les processus et les événements sont créés. Les liens entre les événements et les processus ne sont créés que pendant la simulation sur les points de synchronisation des processus avec le noyau. A l'initialisation, tous les processus, i.e. A et B sont candidats à l'exécution.

Puisque le tirage aléatoire des processus est activé, on peut obtenir plusieurs exécutions suivant que l'on commence la simulation par l'action A ou par l'action B.

Si on commence par A, on exécute A, mais A ne déclenche pas B. Le processus B n'est pas encore sensibilisé à la requête provenant de A. La requête émise par A est perdue. Mais on exécute tout de même B parce que celui-ci est candidat à l'exécution du fait de l'initialisation. B s'exécute jusqu'au point de synchronisation du processus avec le noyau. Il se sensibilise sur la requête provenant de A, puis se place en attente d'une requête en provenance de A. Dans ce cas, il n'y pas de coïncidence entre le déclenchement du processus B du fait de l'occurrence de la requête et son déclenchement du fait de l'initialisation. Cela conduit au comportement suivant :

```
1 pré_calcul_A() ;
2 notification(requête) ;
3 post_calcul_A() ;
4 pré_calcul_B() ;
5 synchronise(requête) ;
```

Si on commence par B, on exécute B à l'initialisation, puis A qui déclenche B, puis B. Cela conduit au comportement suivant :

```
1 pré_calcul_B() ;
2 synchronise(requête) ;
3 pré_calcul_A() ;
4 notification(requête) ;
5 post_calcul_A() ;
6 pré_calcul_B() ;
7 synchronise(requête) ;
8 post_calcul_B() ;
```

De même qu'au paragraphe 2.5.2, la spécification n'est pas respectée. En effet, nous n'avons spécifié qu'un seul comportement et on en obtient toujours deux. Par contre, il n'y a pas de coïncidence fortuite entre l'exécution d'un processus à cause de l'initialisation et son exécution à cause d'une occurrence de requête.

De plus, dans le second cas, non seulement cela ne correspond pas à la spécification, mais les effets de l'exécution de la fonction « pré_calcul_B() » pourraient être indésirables.

2.5.4 Conclusion

Le tirage aléatoire des processus est nécessaire à la sémantique d'exécution des systèmes à événements discrets asynchrones. Sachant que tous les processus sont candidats à l'exécution à l'initialisation, nous voyons que le recours à la sensibilisation statique des processus ne respecte pas la spécification. En fait, elle introduit une confusion entre l'exécution des processus due à l'initialisation et l'exécution des processus due à la sensibilité de ces derniers.

Nous voyons également que le recours à la sensibilisation dynamique produit des comportements multiples. Par contre, il y a un indice permettant de distinguer l'exécution due à l'initialisation, de l'exécution due à la sensibilité des processus. Il s'agit de la perte d'occurrence d'une requête. Dans la suite, on s'appuie sur cette caractéristique pour éliminer les comportements indésirables à l'initialisation.

2.6 Contraintes de modélisation et de simulation

On dérive de cette étude des contraintes de modélisation et de simulation des circuits numériques asynchrones dans le standard SystemC v2.0.1.

Premièrement, il est interdit d'émettre des occurrences immédiates et des occurrences à délai nul du même événement à partir de processus en parallèle communicants dans la même direction sur le même canal.

Deuxièmement, à l'initialisation, tous les processus sont candidats à l'exécution.

Troisièmement, on a recours à la sensibilisation dynamique des processus.

Quatrièmement, si des processus ne produisent pas d'occurrence d'événement à l'initialisation, ils commencent par se synchroniser avec le noyau de simulation pour élaborer les liens entre événements et processus avant tout calcul.

Cinquièmement, si des processus produisent des occurrences d'événement au cours de cette phase de sensibilisation dynamique, ces occurrences doivent être notifiées une fois l'élaboration terminée. Le premier pas de calcul s'effectuant au niveau 0 de l'algorithme de simulation, un moyen de garantir cette relation de causalité consiste à produire ces premières occurrences au niveau 1, i.e. avec une temporisation à délai nul.

Ainsi, l'algorithme de simulation de SystemC v2.0.1 permettant de supporter la sémantique d'exécution des circuits numériques asynchrones se découpe en trois niveaux de calcul :

- Niveau 0 : sensibilisation dynamique des liens entre événements et processus,
- Niveau 1 : temporisation à délai nul,
- Niveau 2 : temporisation à délai non nul.

NB : Le niveau 0 est réservé à la sensibilisation dynamique des processus.

3 Structure d'un modèle de circuit non temporisé dans SystemC v2.0.1

3.1 Choix d'un type de canal de communication

Nous basons la modélisation des circuits numériques asynchrones sur le paradigme de programmation de SystemC v2.0.1 de séparation du calcul et de la communication. Les canaux de communication sont abstraits. Un modèle SystemC v2.0.1 est découpé en modules et en canaux de communication. Ce découpage répond au diagramme de la Figure 18 exposé au paragraphe III3.5.

Il faut noter que SystemC v2.0.1 propose deux types de canaux de communication différents : les canaux élémentaires et les canaux hiérarchiques. Pour saisir ces deux concepts, on peut faire une analogie entre données et communication. Dans un langage où les données sont abstraites, on a des types de données natifs et des types de données définis par l'utilisateur, comme en C++ par exemple. Dans un langage où la communication est abstraite, on a des canaux élémentaires et des canaux hiérarchiques, une des caractéristiques de SystemC v2.0.1.

Les canaux élémentaires correspondraient aux types de données natifs alors que les canaux hiérarchiques correspondraient aux types de données définis par l'utilisateur.

Pour des raisons de souplesse, de puissance d'utilisation et par souci de compatibilité avec l'environnement SystemC v2.0.1 existant, nous nous focalisons sur les canaux hiérarchiques. Nous laissons les canaux élémentaires et la mécanique de simulation associée intactes.

Nous présentons ensuite un canal de communication hiérarchique point à point unidirectionnel supportant deux modes : un mode direct et un mode à poignée de main et des modules permettant de composer des circuits booléens. Cette structure constitue le squelette de notre modèle de circuits numériques asynchrones. Elle prend en compte la mise à jour des valeurs au niveau des canaux de communication en reposant sur le mécanisme de production-consommation des événements présenté au paragraphe 2.1.

3.2 Canal de communication point à point unidirectionnel direct

3.2.1 Interface d'entrée du canal

L'interface d'entrée du canal spécifie le protocole de communication entre le port de sortie d'un module et l'entrée du canal. Le protocole de communication entre les modules est point à point, unidirectionnel et direct. Au niveau de l'entrée du canal ou d'un port de sortie d'un module, il consiste en la production d'une requête accompagnée de l'écriture d'une nouvelle valeur dans le canal. Voici son pseudo code.

Pseudo code de l'interface d'entrée du canal :

```
1 interface_entrée_canal_sortie_port {
2     // méthodes
3     produit_requête() ;
4     écrit_valeur(booléen valeur) ;
5     // variables
6     événement requête ;
7     booléen valeur_entrée ;
8 }
```

3.2.2 Interface de sortie du canal

L'interface de sortie du canal spécifie le protocole de communication entre la sortie du canal et le port d'entrée d'un module. Le protocole de communication entre les modules est point à point, unidirectionnel et direct. Au niveau de la sortie du canal ou d'un port d'entrée d'un module, il consiste en la consommation d'une requête, en la mise à jour et en la lecture d'une nouvelle valeur depuis le canal. Voici son pseudo code.

Pseudo code de l'interface de sortie du canal :

```
1 interface_sortie_canal_entrée_port {
2     // méthodes
3     consomme_requête() ;
4     met_à_jour_valeur_canal() ;
5     booléen lit_valeur() ;
6
```

```
7     // variables
8     booléen valeur_sortie;
9  }
```

3.2.3 Canal de communication point à point unidirectionnel direct

Le canal de communication point à point unidirectionnel direct implémente le protocole de communication point à point unidirectionnel direct spécifié par les interfaces d'entrée et de sortie du canal (voir Figure 24).

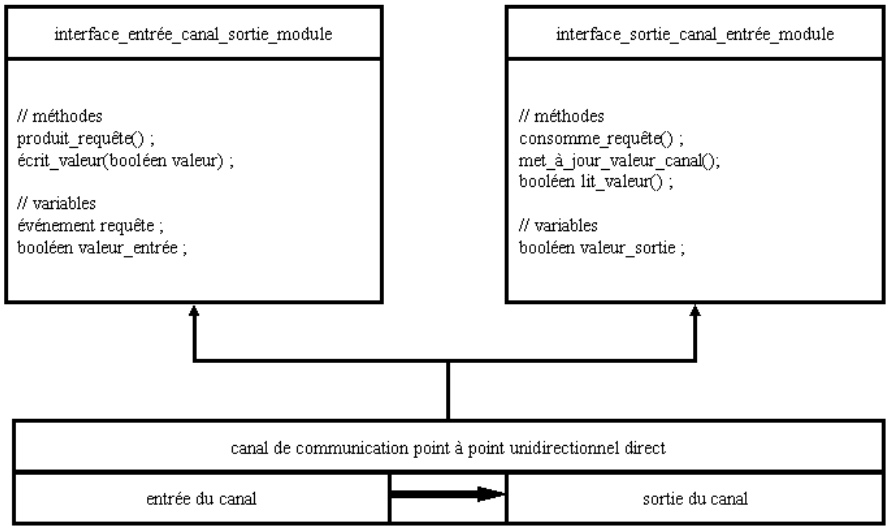


Figure 24 : Canal de communication point à point unidirectionnel direct

Le pseudo code du canal de communication point à point unidirectionnel direct est développé à l'annexe C.3.

3.3 Canal de communication point à point unidirectionnel à poignée de main

3.3.1 Introduction

On construit à partir du canal de communication point à point unidirectionnel direct, un canal de communication point à point unidirectionnel à poignée de main. Ce complément s'effectue sans modifier la relation entre la requête et le changement de valeur. On ajoute dans le protocole un acquittement et les méthodes associées. On pourra noter que la communication reste unidirectionnelle.

Dans le cas des circuits numériques asynchrones, cela sera bien utile pour modéliser l'environnement et initialiser la simulation comme nous le verrons au paragraphe 6.

Le concepteur sera ensuite libre d'utiliser le canal dans le mode direct ou bien dans le mode à poignée de main suivant la fonctionnalité à décrire.

3.3.2 Interface d'entrée du canal

L'interface d'entrée du canal est complétée. Spécifiant un protocole de communication point à point unidirectionnel direct entre l'entrée du canal et un port de sortie d'un module, elle supporte maintenant le protocole à poignée de main. Voici son pseudo code.

Pseudo code de l'interface d'entrée du canal :

```
1 interface_entrée_canal_sortie_port {
2     // méthodes
3     produit_requête() ;
4     consomme_acquittement() ;
5     écrit_valeur(booléen valeur) ;
6     // variables
7     événement requête ;
8     booléen valeur_entrée ;
9 }
```

3.3.3 Interface de sortie du canal

L'interface de sortie du canal est complétée. Spécifiant un protocole de communication point à point unidirectionnel direct entre la sortie du canal et un port d'entrée d'un module, elle supporte maintenant le protocole à poignée de main. Voici son pseudo code.

Pseudo code de l'interface de sortie du canal :

```
1 interface_sortie_canal_entrée_port {
2     // méthodes
3     produit_acquittement() ;
4     consomme_requête() ;
5     met_à_jour_valeur_canal() ;
6     booléen lit_valeur() ;
7     // variables
8     événement acquittement ;
9     booléen valeur_sortie;
10 }
```

3.3.4 Canal de communication point à point unidirectionnel à poignée de main

Le canal de communication point à point unidirectionnel à poignée de main implémente le protocole de communication point à point unidirectionnel à poignée de main spécifié par les interfaces d'entrée et de sortie du canal (voir Figure 25).

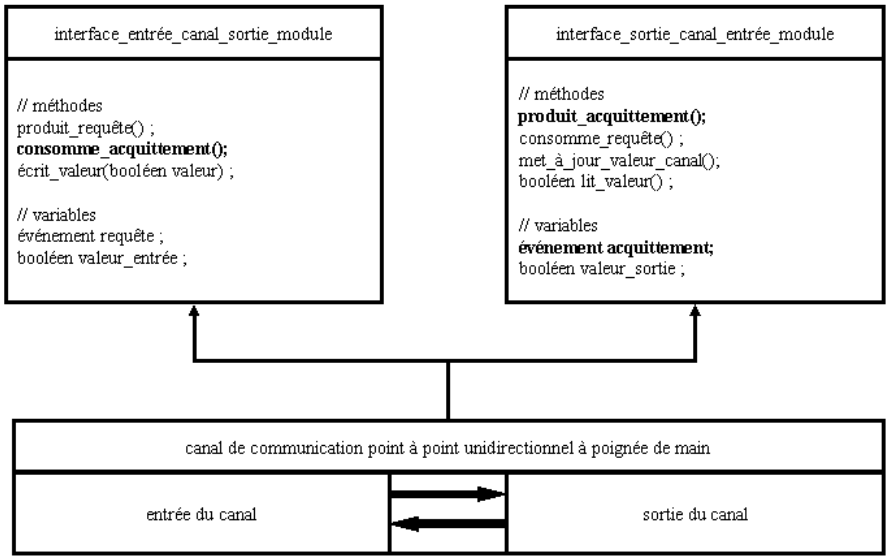


Figure 25 : Canal de communication point à point unidirectionnel à poignée de main

Le pseudo code du canal de communication point à point unidirectionnel à poignée de main est développé à l'annexe C.4.

3.4 Modules

3.4.1 Introduction

Dans SystemC v2.0.1, un module encapsule un ou plusieurs ports d'entrée, un ou plusieurs ports de sortie et un ou plusieurs processus. Les ports possèdent un type défini par l'interface de communication qu'ils utilisent. Ce type définit le protocole de communication du module à travers ce port. Dans notre cas, les ports utilisent les interfaces définies au paragraphe 3.3.

Nous allons présenter ces modules en commençant par le plus simple : le module mono entrée, mono sortie et mono processus. Nous verrons ensuite comment généraliser ce module pour obtenir plusieurs ports d'entrée ou de sortie, ou plusieurs processus. Ces différents axes de généralisation sont indépendants. Il suffira de composer ces différents points en fonction du besoin.

3.4.2 Module mono entrée, mono sortie, mono processus

Typiquement, la fonctionnalité implémentée par un module mono entrée, mono sortie, mono processus (voir Figure 26) consiste en une synchronisation avec l'environnement, une lecture du port d'entrée, un calcul et une écriture sur le port de sortie.

La lecture consiste en trois phases : la consommation de l'événement, la mise à jour de la valeur du canal d'entrée et la lecture de la valeur du canal d'entrée.

Au niveau du calcul, on dispose de tous les motifs permis par le recours au langage C++, y compris le recours à des variables déclarées au niveau du module.

L'écriture consiste en deux phases : la production d'un événement sur le canal de sortie et l'écriture d'une nouvelle valeur sur le canal de sortie.

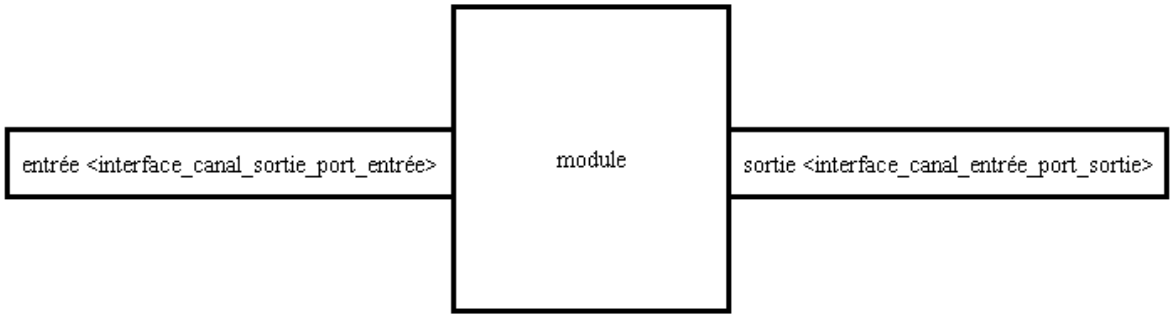


Figure 26 : Module mono entrée, mono sortie, mono processus

Le pseudo code du module mono entrée, mono sortie, mono processus est développé à l'annexe C.5.

3.4.3 Module multi entrées, mono sortie, mono processus

Du côté de l'entrée, SystemC v2.0.1 fournit plusieurs motifs de synchronisation comme on l'a vu au paragraphe III3.4.6. Notamment, il est possible de déclencher un processus sur une disjonction d'occurrences d'événements. Cette fonctionnalité est nécessaire lorsque le module héberge plusieurs ports d'entrée comme sur la Figure 27. Il faut alors modifier la synchronisation du processus en conséquence. Seules les valeurs d'entrée ayant changées doivent être prises en compte dans le nouveau calcul.

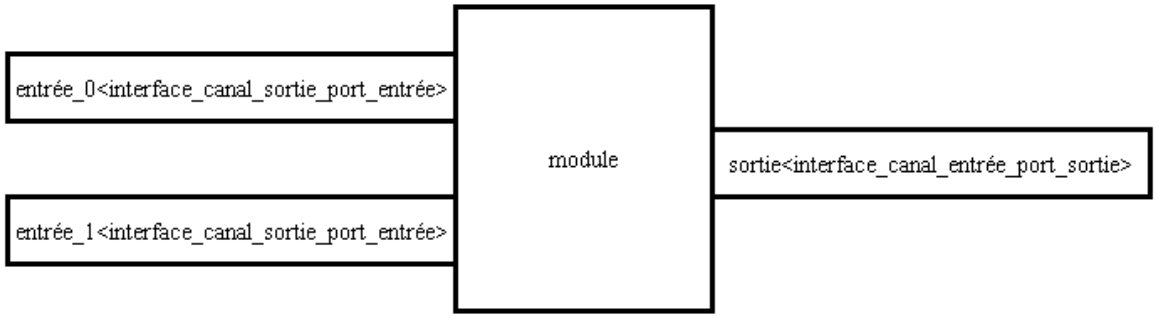


Figure 27 : Module bi entrées, mono sortie, mono processus

Le pseudo code du module bi entrées, mono sortie, mono processus à l'annexe C.6.

3.4.4 Module mono entrée, multi sorties, mono processus

Du côté de la sortie, l'échange d'information à travers un port s'effectue nécessairement sur un point de synchronisation du modèle avec le noyau de simulation. Ainsi, lorsqu'au sein d'un processus, on procède à plusieurs notifications d'occurrences d'événements sur des ports de sortie différents, bien que ces notifications d'événement soient programmées en série en C++, la synchronisation suivante avec l'environnement induit leur interprétation en parallèle. La modélisation du parallélisme dans SystemC v2.0.1 s'effectue donc par une série de notifications suivie d'une synchronisation avec le noyau de simulation. La Figure 28 représente un module mono entrée, bi sorties, mono processus.

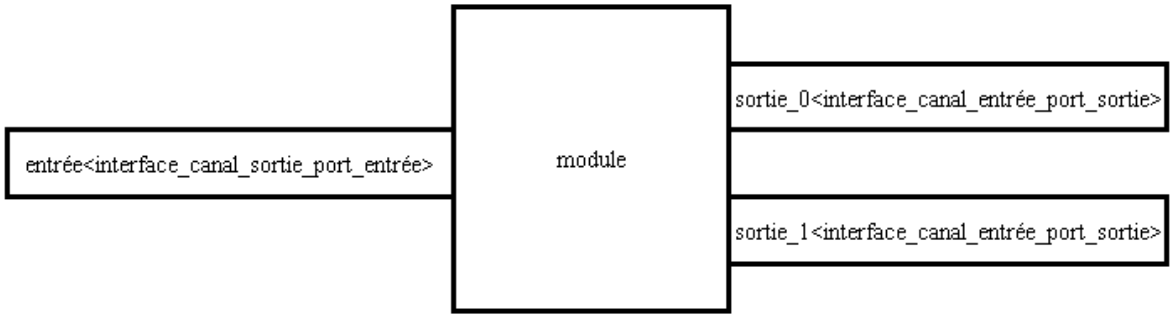


Figure 28 : Module mono entrée, bi sorties, mono processus

Le pseudo code du module mono entrée, bi sorties, mono processus à l'annexe C.7

3.4.5 Module mono entrée, mono sortie, multi processus

Par ailleurs, un module peut contenir plusieurs processus. Ces processus sont exécutés en parallèle suivant le principe des co-routines présenté au chapitre III. Suivant ce principe, les processus sont entrelacés et simulés les uns après les autres. L'élaboration fait apparaître une fourche implicite à l'entrée du module. Chaque processus partage les variables du module dont le port de sortie ; ce qui fait apparaître un point de contention à la sortie (voir Figure 29).

Les processus sont mis à plat avec les autres processus du système au moment de l'élaboration. Dans le cas où il y a des variables partagées au niveau du module, il faut identifier la fin de l'exécution, i.e. la fin de l'exécution de tous les processus du module. On utilise un compteur de processus. On met cette caractéristique en œuvre au chapitre V pour la vérification de la propriété de sûreté.

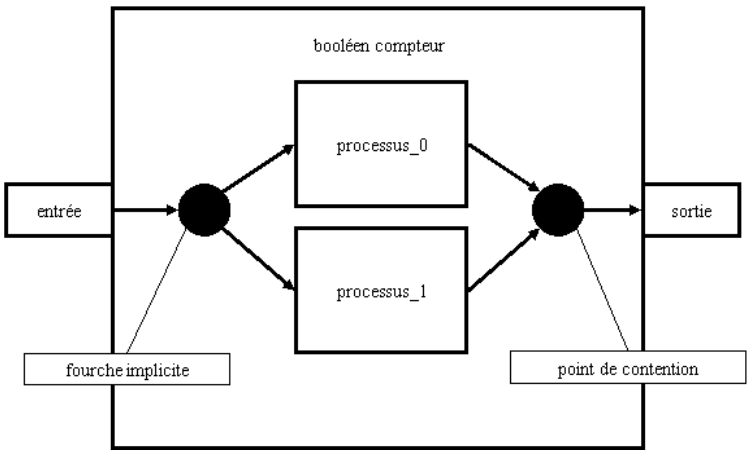


Figure 29 : Module mono entrée, mono sortie, bi processus

Le pseudo code du module mono entrée, mono sortie, bi processus est développé à l'annexe C.8.

3.5 Composition d'un circuit

On dispose alors de nombreux éléments pour assembler un circuit, ce circuit pouvant lui-même être encapsulé dans un module. Parmi ces éléments, on trouve un canal de

communication point à point unidirectionnel que l'on peut utiliser soit en mode direct, soit en mode à poignée de main. Ce canal prend en charge la mise à jour des valeurs.

On trouve également des modules. Ils peuvent posséder un ou plusieurs ports d'entrée, un ou plusieurs ports de sortie, un ou plusieurs processus. Les types utilisés par les ports des modules sont les interfaces définissant le protocole de communication implémenté par le canal sus cité.

Pour créer un circuit, on déclare des modules et des canaux et on interconnecte les modules par les canaux via les ports. La Figure 30 représente un exemple de circuit.

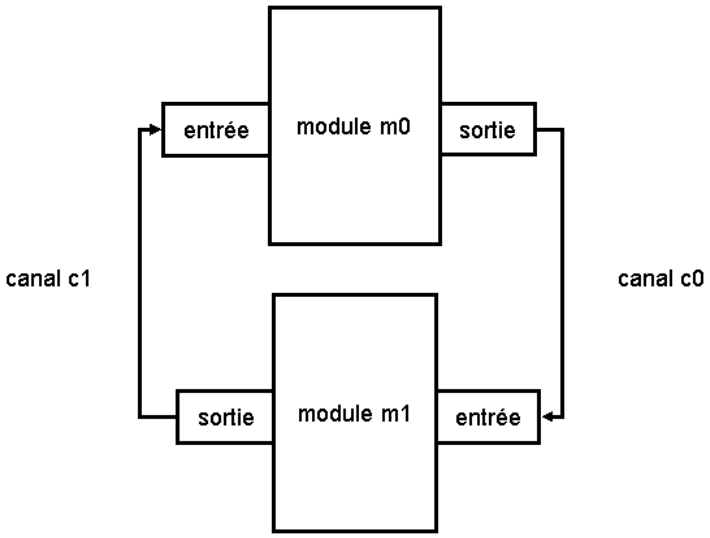


Figure 30 : Composition d'un circuit basé sur le paradigme de programmation calcul versus communication de SystemC v2.0.1

Le pseudo code du circuit basé sur le paradigme de programmation calcul versus communication est développé à l'annexe C.9.

4 Formalisation des éléments des circuits quasi insensibles aux délais

L'objectif de cette section est la formalisation des circuits quasi insensibles aux délais en LTL. L'intérêt d'une telle formalisation réside en plusieurs points. D'abord, cela permet de structurer notre réflexion autour des circuits numériques asynchrones. Ensuite, cela permet de préparer l'analyse de l'interopérabilité entre certains outils de vérification formelle et SystemC v2.0.1. Le choix de ce cadre mathématique est guidé par plusieurs critères : les aspects syntaxiques et les aspects sémantiques de ces circuits d'une part, le modèle de temps de SystemC v2.0.1, d'autre part.

Dans un premier temps, nous présentons briévement LTL. Dans un second temps, nous formalisons les circuits quasi insensibles aux délais dans ce cadre.

4.1 Logique temporelle linéaire

4.1.1 Introduction

La logique temporelle linéaire (LTL) est née dans les années 1970 des travaux de Amir Pnueli. On en trouve une présentation très complète dans [26]. Les logiques temporelles ont largement évolué depuis lors. Elles s'inscrivent dans le cadre de la logique polymodale où l'on retrouve le concept de bisimulation dérivé de CCS, mais cela sort du cadre de cette thèse.

Elle fournit un cadre mathématique adequat pour la formalisation des circuits quasi insensibles aux délais présentés au paragraphe II3. En effet, d'une part, les éléments des circuits quasi insensibles aux délais se traduisent presque directement dans ce cadre. D'autre part, on se limite à un modèle de temps linéaire et discret, les états du système étant indexés par les entiers naturels. Ce modèle de temps est donc très proche du modèle de temps de SystemC v2.0.1.

On propose donc de modéliser ici les éléments d'un circuit quasi insensible aux délais en LTL. On se reportera au chapitre V pour la formalisation des propriétés d'insensibilité aux délais dans le même cadre.

4.1.2 Présentation

On peut fournir une formalisation de la sémantique des opérateurs $\neg$ (« non »), $\vee$ (« ou ») et U (« jusqu'à », « until » en anglais) comme suit.

Etant donné un modèle de Kripke [26], une suite d'états dans le temps noté : $\sigma : s_0, s_1, \ldots$, on définit la notion de formule temporelle p vraie à un instant $j \geq 0$ dans σ par $(\sigma, j) \models p$,

- $(\sigma, j) \models p \Leftrightarrow s_j \models p$ correspond à l'évaluation d'une formule p à partir d'un état s_j,
- $(\sigma, j) \models \neg p \Leftrightarrow (\sigma, j) \not\models p$,
- $(\sigma, j) \models p \vee q \Leftrightarrow (\sigma, j) \models p$ ou $(\sigma, j) \models q$,
- $(\sigma, j) \models pUq \Leftrightarrow$ il existe $k \geq j, (\sigma, k) \models q$ et pour tout i tel que $j \leq i < k, (\sigma, i) \models p$.

Cela implique la sémantique suivante pour les opérateurs dérivés $\Diamond$ (lire « inéluctablement ») et $\Box$ (lire « pour toujours »):

- $(\sigma, j) \models \Diamond p \Leftrightarrow$ il existe $k \geq j, (\sigma, k) \models p$,
- $(\sigma, j) \models \Box p \Leftrightarrow$ pour tout $k \geq j, (\sigma, k) \models p$.

4.2 Règle de production à une seule sortie

La règle de production à une seule sortie a été présentée au paragraphe II3.5. L'opérateur U permet de spécifier une règle de production à une seule sortie assortie de la contrainte de complétion de la transition cible, qu'elle soit montante (premier point) ou descendante (second point) :

- $p \rightarrow q \uparrow \Leftrightarrow (\sigma, j) \models pUq$,
- $p \rightarrow q \downarrow \Leftrightarrow (\sigma, j) \models pU\neg q$.

4.3 Règle de production à plusieurs sorties

La règle de production à plusieurs sorties a été présentée au paragraphe II3.7. L'opérateur U permet aussi de spécifier une règle de production à plusieurs sorties assortie de la contrainte de complétion de toutes les transitions cibles :

- $p \to q_0 \uparrow, q_1 \uparrow \Leftrightarrow (\sigma, j) \models pU(q_0 \wedge q_1)$

NB : Rien n'impose a priori que les deux variables de sortie deviennent vraies au même instant. Il suffit qu'au bout d'un certain temps les deux soient vraies.

4.4 Porte à une seule sortie

La porte à une seule sortie a été présentée au paragraphe II3.6. Elle s'obtient par la donnée de deux règles de production à une seule sortie dont les transitions cibles sont opposées :

- $\begin{cases} p_0 \to q \uparrow \Leftrightarrow (\sigma, j) \models p_0 U q \\ p_1 \to q \downarrow \Leftrightarrow (\sigma, j) \models p_1 U \neg q \end{cases}$

4.5 Porte « fourche »

La fourche a été présentée au paragraphe II3.8. Elle s'obtient par la donnée de deux règles de production à plusieurs sorties dont les transitions cibles sont opposées :

- $\begin{cases} p \to q_0 \uparrow, q_1 \uparrow, ... \Leftrightarrow (\sigma, j) \models pU(q_0 \wedge q_1 \wedge ...) \\ \neg p \to q_0 \downarrow, q_1 \downarrow, ... \Leftrightarrow (\sigma, j) \models \neg pU(\neg q_0 \wedge \neg q_1 \wedge ...) \end{cases}$

5 Application aux circuits quasi insensibles aux délais

5.1 Modélisation de la règle de production à une seule sortie

La règle de production a été formalisée en LTL au paragraphe 4.2. On associe un module multi entrées, mono sortie, mono processus à une règle de production. Ce module doit posséder autant de ports d'entrée que de variables booléennes dans la garde de la règle de production et un seul port de sortie.

On considère ici un événement pour déclencher l'exécution de la règle et un événement pour marquer la fin de la transition. Sachant que le noyau de simulation n'est pas préemptif (voir paragraphe III3.4.7), cela revient à regrouper le calcul de la garde et le calcul de la transition dans la même action atomique.

On peut également voir une règle de production comme une commande gardée. Le calcul de la transition est conditionné par la valeur issue du calcul de la garde. L'émission de l'occurrence d'événement marquant la fin de la transition est donc conditionnelle.

Dans le cas général, le processus est déclenché sur une disjonction d'occurrences d'événements véhiculés par les différents canaux d'entrée. On représente dans le pseudo code suivant le cas particulier de la règle de production : $p \to q \uparrow \Leftrightarrow (\sigma, j) \models pUq$ qui donne un module avec un seul port d'entrée et un seul port de sortie.

Pseudo code du modèle SystemC v2.0.1 d'une règle de production :

```
1  module {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée ;
4      port <interface_entrée_canal_sortie_port> sortie ;
5      // variables
6      booléen valeur, garde ;
7      // processus
8      processus {
9            // synchronisation du processus
10           synchronise( entrée.requête ) ;
11           // lecture
12           entrée.consomme_requête() ;
13           valeur = entrée.lit_valeur();
14           // calcul de la garde
15           garde = calcul(valeur) ;
16           si garde = vrai {
17                 // écriture
18                 sortie.produit_requête() ;
19                 sortie.écrit_valeur(vrai) ;
20           }
21     }
22 }
```

On peut affiner la modélisation en prenant en compte l'effectivité des transitions décrites au paragraphe II3.9 dans le modèle des circuits quasi insensibles aux délais. En effet à quoi bon propager un nouvel événement dans le circuit si la valeur ne change pas ?

L'écriture devient non seulement conditionnelle en fonction de la garde, mais aussi en fonction de la valeur d'entrée du canal du port de sortie. On signale le cas échéant au concepteur qu'une transition ineffective est intervenue. La manière dont le concepteur peut tirer profit de cette information pour améliorer les performances de son modèle sort du cadre de cette thèse.

Pseudo code du modèle SystemC v2.0.1 d'une règle de production avec prise en compte de l'effectivité des transitions :

```
1  module {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée ;
4      port <interface_entrée_canal_sortie_port> sortie ;
5      // variables
6      booléen valeur, garde ;
7      // processus
```

```
8     processus {
9           // synchronisation du processus
10          synchronise( entrée.requête ) ;
11          // lecture
12          entrée.consomme_requête() ;
13          valeur = entrée.lit_valeur();
14          // calcul
15          garde = calcul() ;
16          si garde = vrai {
17                // écriture
18                si sortie.valeur_entrée != vrai {
19                      sortie.produit_requête() ;
20                      sortie.écrit_valeur(vrai) ;
21                }
22                sinon
23                      détecte_transition_ineffective() ;
24          }
25    }
26 }
```

5.2 Modélisation de la porte à une seule sortie

La porte à une seule sortie a été formalisée en LTL au paragraphe 4.4. On associe un module bi-processus à une porte à une seule sortie. Ce module doit posséder autant de ports d'entrée que de variables booléennes dans l'union des variables des deux gardes et un seul port de sortie.

On considère systématiquement que le processus n°0 traite la transition cible descendante et que le processus n°1 traite la transition cible montante.

Comme pour la règle de production, dans le cas général, le processus est déclenché sur une disjonction d'occurrences d'événements véhiculés par les différents canaux d'entrée. Mais par souci de clarté, on ne représente ici qu'un seul port d'entrée.

Pseudo code du modèle SystemC v2.0.1 d'une porte à une seule sortie :

```
1  module {
2     // ports
3     port <interface_sortie_canal_entrée_port> entrée ;
4     port <interface_entrée_canal_sortie_port> sortie ;
5     // compteur de processus initialisé à 0
6     entier compteur ;
7     // variables du processus 0
8     booléen valeur_0, garde_0 ;
9     // processus 0
10    processus_0 {
```

```
11          // synchronisation du processus 0
12          synchronise( entrée.requête ) ;
13          // lecture
14          entrée.consomme_requête() ;
15          valeur_0 = entrée.lit_valeur();
16          // calcul
17          garde_0 = calcul(valeur_0) ;
18          si garde_0 = vrai {
19                // écriture
20                si sortie.valeur_entrée != faux {
21                      sortie.produit_requête() ;
22                      sortie.écrit_valeur(faux) ;
23                }
24                sinon
25                      détecte_transition_ineffective() ;
26          }
27          // détection de la fin de l'exécution du module
28          compteur = compteur + 1 ;
29          si compteur = 2
30                compteur = 0 ;
31    }
32    // variables du processus 1
33    booléen valeur_1, garde_1 ;
34    // processus 1
35    processus_1 {
36          // synchronisation du processus 1
37          synchronise( entrée.requête ) ;
38          // lecture
39          entrée.consomme_requête() ;
40          valeur_1 = entrée.lit_valeur();
41          // calcul
42          garde_1 = calcul(valeur_1) ;
43          si garde_1 = vrai {
44                // écriture
45                si sortie.valeur_entrée != vrai {
46                      sortie.produit_requête() ;
47                      sortie.écrit_valeur(vrai) ;
48                }
49                sinon
50                      détecte_transition_ineffective() ;
```

```
51          }
52          // détection de la fin de l'exécution du module
53          compteur = compteur + 1 ;
54          si compteur = 2
55                compteur = 0 ;
56    }
57 }
```

5.3 Exemple de modélisation de porte bi entrées, mono sortie : la porte de Muller

La porte de Muller [11] est une porte à deux entrées et à une seule sortie spécifique en logique asynchrone. C'est l'élément qui assure la synchronisation locale entre deux signaux. Dès que les deux signaux d'entrée sont vrais, elle produit une transition montante sur la sortie. Dès que les deux signaux d'entrée sont faux, elle produit une transition descendante sur la sortie. Sinon, rien ne se passe.

Son expression sous forme de règles de production est la suivante :

- $\begin{cases} p_0 \wedge p_1 \rightarrow q\uparrow \\ \neg p_0 \wedge \neg p_1 \rightarrow q\downarrow \end{cases}$.

En LTL, cela donne :

- $\begin{cases} (\sigma, j) \models p_0 \wedge p_1 U q \\ (\sigma, j) \models \neg p_0 \wedge \neg p_1 U \neg q \end{cases}$.

La différence principale avec l'exemple de la porte à une seule sortie est la présence de plusieurs canaux en entrée.

La Figure 31 montre la représentation graphique d'une porte de Muller sous forme de réseau de Petri, le symbole couramment utilisé pour cet élément et une implémentation possible sous forme de circuit CMOS.

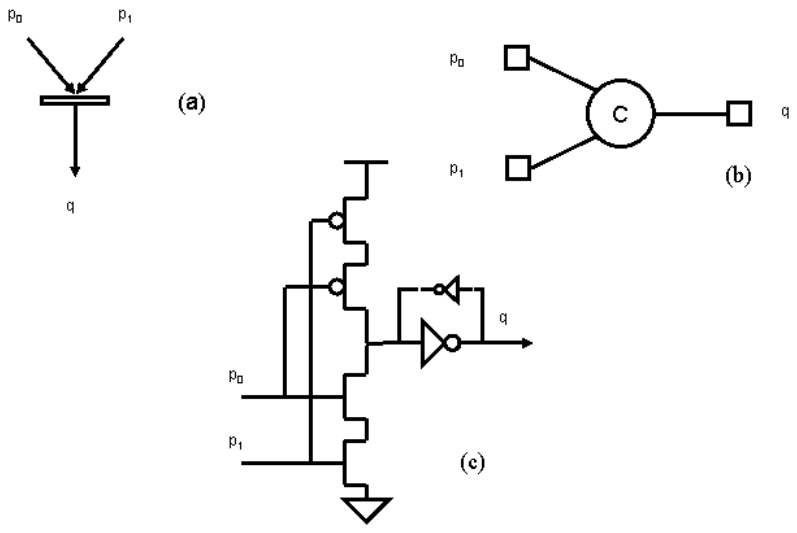


Figure 31 : Porte de Muller : réseau de Petri ordinaire (a), symbole (b) et circuit CMOS (c).

Le pseudo code du modèle de la porte de Muller est développé à l'annexe C.11.4.

NB : Il n'y a jamais de conflit d'écriture sur la sortie car les gardes sont statiquement mutuellement exclusives.

5.4 Modélisation de la fourche

La fourche a été formalisée en LTL au paragraphe 4.5. C'est une porte spécifique avec une seule entrée et deux ou plusieurs sorties. C'est la seule porte à plusieurs sorties autorisée dans le modèle quasi insensible aux délais. Dans le contexte de SystemC v2.0.1, la modélisation du parallélisme repose sur le concept de co-routines présenté au chapitre III.

Son expression sous forme de règles de production est la suivante (on se limite au cas à deux sorties) :

- $\begin{cases} p \to q_0 \uparrow, q_1 \uparrow \\ \neg p \to q_0 \downarrow, q_1 \downarrow \end{cases}$.

En LTL, cela donne :

- $\begin{cases} (\sigma, j) \models pU\left(q_0 \wedge q_1\right) \\ (\sigma, j) \models \neg pU\left(\neg q_0 \wedge \neg q_1\right) \end{cases}$.

La Figure 32 montre la représentation graphique d'une fourche sous forme de réseau de Petri, le symbole couramment utilisé pour cet élément et une implémentation possible (circuit CMOS à base d'inverseurs).

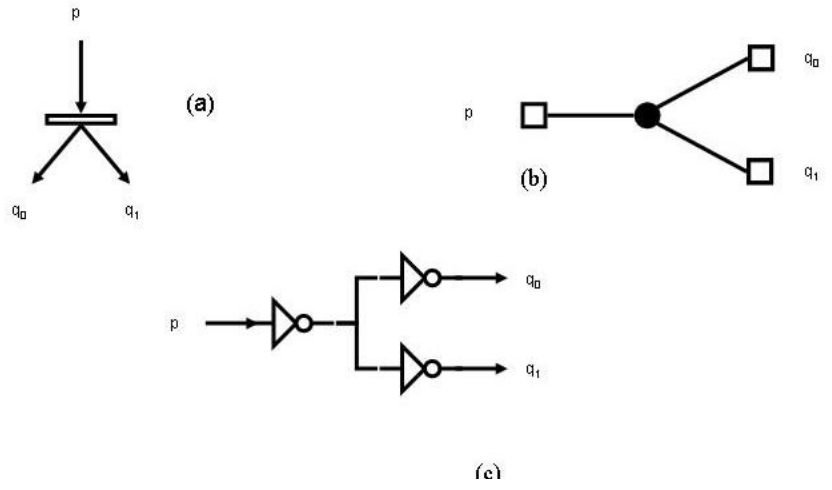


Figure 32 : Fourche. Réseau de Petri ordinaire (a), symbole (b) et circuit CMOS (c).

Le pseudo code du modèle de la fourche est développé à l'annexe C.11.2.

NB : Il n'y a jamais de conflit d'écriture sur la sortie car les gardes sont statiquement mutuellement exclusives.

6 Initialisation et éléments d'environnement

6.1 Introduction

Dans un circuit numérique synchrone, il y a toujours une ou plusieurs horloges chargées de cadencer l'évolution du circuit. Du coup, la question du démarrage de la simulation ne se pose pas de manière aussi cruciale que dans le cas d'un circuit numérique asynchrone. En effet, dans ce dernier cas, la synchronisation est locale et le couplage entre données et synchronisation est fort. Il y a dans un tel circuit nécessairement un ou plusieurs éléments qui vont déclencher le ou les premiers événements dans le circuit.

Les éléments que nous avons décrits jusque-là, règle de production, porte et fourche se bloquent dès l'initialisation puisqu'ils commencent par se synchroniser avec le noyau de simulation en attente d'un événement. Ces éléments ne peuvent donc pas être chargés de produire le ou les premiers événements de la simulation. Dans un circuit donné, nous pourrions identifier les instances de ces éléments chargés du démarrage et adapter leur comportement en conséquence. Mais le modèle perdrait en régularité.

Nous choisissons de conserver le motif régulier du comportement développé jusqu'ici. Ce motif consiste à d'abord synchroniser un élément du circuit avec l'environnement avant de procéder au calcul. Nous introduisons alors dans le modèle un élément spécifique dont la tâche sera de produire un premier événement avant de se synchroniser avec le noyau. Ce sera le seul élément du modèle dont le comportement commencera par un calcul et une écriture sur un canal avant de se synchroniser avec son environnement. On appelera cet élément le « producteur ». Par exemple, cet élément pourra servir de générateur de mise à zéro.

Enfin, par symétrie, pour des raisons de confort d'utilisation et pour compléter le modèle, on introduira un élément nommé « consommateur », élément dual du producteur.

6.2 Producteur

Par définition, le producteur ne possède pas de port d'entrée. C'est un module hiérarchique (voir paragraphe III3.5 pour la notion de hiérarchie en SystemC v2.0.1) qui se décompose en deux modules : le générateur et l'adaptateur (voir Figure 33). Ces deux modules mono entrée, mono sortie et mono processus communiquent suivant le protocole point à point unidirectionnel à poignée de main décrit paragraphe 3.3. Le générateur envoie une requête à l'adaptateur, puis se synchronise sur sa sortie et attend un acquittement. L'adaptateur se synchronise sur son entrée et renvoie un acquittement au générateur. Ensuite, l'adaptateur écrit sur sa sortie suivant le protocole point à point unidirectionnel direct. L'adaptateur réalise ainsi l'adaptation entre le protocole à poignée de main en entrée et le protocole direct en sortie, d'où son nom.

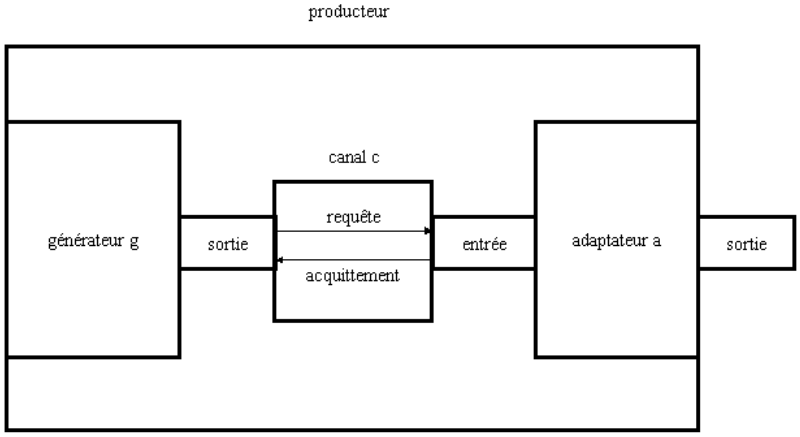


Figure 33 : Diagramme du modèle du producteur

Le pseudo code du modèle du producteur est développé à l'annexe C.11.1.

6.3 Consommateur

Par définition, le consommateur ne possède pas de port de sortie. Sa modélisation ne pose pas de problème de synchronisation particulier. Le protocole de communication point à point unidirectionnel direct suffit. Il est représenté sur le diagramme de la Figure 34.

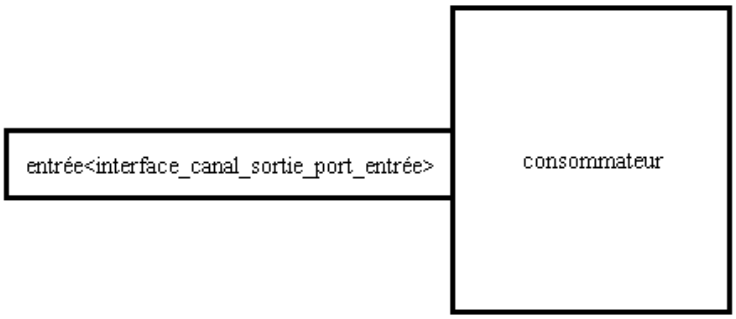


Figure 34 : Diagramme du modèle du consommateur

Le pseudo code du modèle du consommateur est développé à l'annexe C.10.

7 Temporisation

7.1 Composition

Nous avons vu au paragraphe III3.4.7 qu'il existait trois niveaux de simulation dans l'algorithme d'ordonnancement de SystemC v2.0.1. Le niveau 0 correspond aux notifications immédiates, le niveau 1 aux notifications à délai nul et le niveau 2 aux notifications à délai non nul.

Le niveau 0 possède une spécificité. A ce niveau, les processus sont exécutés dynamiquement au sein de la même boucle. Ce mécanisme correspond à un ordonnancement que l'on ne retrouve pas au niveau 1, ni au niveau 2. Cet ordonnancement de niveau 0 avec évaluation

dynamique des processus, couplé au tirage aléatoire permet d'obtenir tous les entrelacements possibles des actions à exécuter en parallèle.

Ce n'est pas le cas pour les niveaux 1 et 2. Par contre ces niveaux respectent le même type d'ordonnancement, donc le même comportement, à l'abstraction des délais près. On peut se baser sur ces 2 niveaux pour temporiser le modèle en cohérence avec le comportement simulé.

En écartant l'ordonnancement de niveau 0, on se prive d'un certain nombre d'entrelacements au bénéfice de la temporisation du modèle de simulation. Se prive-t-on de comportements spécifiés par le concepteur ?

On prend en compte dans la modélisation au chapitre V la vérification du partage des ressources entre les processus à travers l'instrumentation de la vérification des propriétés d'insensibilité aux délais. Les comportements dont on se prive correspondent à des processus strictement parallèles, i.e. non communicants.

C'est un argument supplémentaire qui confirme les contraintes de modélisation et de simulation définies au paragraphe 2.6.

Ainsi, l'algorithme de simulation de SystemC v2.0.1 permettant de supporter la sémantique d'exécution des circuits numériques asynchrones se découpe en trois niveaux de calcul :

- Niveau 0 : sensibilisation dynamique des processus,
- Niveau 1 : temporisation à délai nul,
- Niveau 2 : temporisation à délai non nul.

La temporisation est cohérente avec le type d'ordonnancement du niveau 1 et du niveau 2 , i.e. sans perte de comportement significatif dans le passage d'un niveau à l'autre.

7.2 Durée abstraite

Dans les systèmes à événements discrets, on a vu que les événements étaient atomiques, distinguables et de durée nulle. On a donc nécessairement besoin de deux événements pour représenter une durée, l'un marquant le début et l'autre la fin d'une action. L'intervalle entre les occurrences de ces deux événements représente alors une durée, ces occurrences se matérialisant nécessairement lors d'une synchronisation avec le noyau de simulation.

Pseudo code d'un modèle SystemC v2.0.1 illustrant la notion de durée abstraite :

```
1  module {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée ;
4      port <interface_entrée_canal_sortie_port> sortie ;
5      // processus
6      processus {
7            // synchronisation du processus
8            synchronise( entrée.requête ) ;
9            // lecture
10           entrée.consomme_requête() ;
```

```
11          ////////////////////////////////////////
12          //              Durée abstraite           //
13          ////////////////////////////////////////
14          // écriture
15          sortie.produit_requête() ;
16    }
17 }
```

Cette durée est arbitraire dans le cas d'une simulation non temporisée au cours de laquelle les occurrences du même événement sont séparées l'une de l'autre par un ou plusieurs pas de simulation à délai nul au niveau 1 entre lesquels peuvent s'intercaler des pas de simulation de niveau 0.

Ces pas de simulation de niveau 0 peuvent provenir d'un autre modèle de calcul composé avec un circuit numérique asynchrone. Nous prenons donc ce mode non temporisé en compte dans notre modèle de temporisation. Cela permet par la suite de mettre en évidence la rupture de la relation de causalité entre les processus lors de l'initialisation.

On modifie l'interface de communication point à point unidirectionnelle d'entrée de canal pour que la production d'occurrence d'événements prenne en compte ces deux niveaux de simulation non temporisée.

Pseudo code de l'interface d'entrée du canal :

```
1 interface_entrée_canal_sortie_port {
2     // méthodes
3     produit_requête(entier niveau_temporisation) ;
4     consomme_acquittement() ;
5     écrit_valeur(booléen valeur) ;
6     // variables
7     événement requête ;
8     booléen valeur_entrée ;
9 }
```

On modifie le canal de communication point à point unidirectionnel en conséquence.

Pseudo code de la méthode de production de requête du canal :

```
1 // interface_entrée_canal_sortie_port
2 produit_requête(entier niveau_temporisation) {
3     si niveau_temporisation = 0
4           requête.notifie() ;
5     si niveau_temporisation = 1
6           requête.notifie(délai_nul) ;
7 }
```

7.3 Durée concrète

Il existe une très large gamme de modèles « analogiques » de délais pour les circuits intégrés CMOS. Ce type de modélisation est un champ à part entière (enseignement, recherche,

industrie) dont la présentation sort largement du cadre de cette thèse. Ce domaine fut initié dès 1948 par W. C. Elmore [70].

Aujourd'hui, ces modèles réalisent une abstraction, parfois stochastique des paramètres électriques des portes et des fils d'interconnexion. Ces modèles sont soit analytiques, soit linéaires par morceau, soit échantillonnés et stockés dans des tables. A priori, tous ces modèles « analogiques » de délais (en fonction de l'impédance d'entrée, de la transconductance, de l'impédance de sortie pour les portes ou bien C, RC, RLC pour les fils) peuvent être modélisés dans SystemC v2.0.1. Il s'agit de paramétrer les portes et les canaux de communication convenablement.

Dans un flot classique de conception de circuits intégrés, la tâche de l'évaluation des délais est prise en charge par un extracteur de composants parasites, puis par un calculateur de délais à partir des informations extraites du dessin des masques.

Dans notre cas, puisque l'exécution du modèle SystemC v2.0.1 d'une règle de production est atomique, on se contente de temporiser les canaux de communication du circuit.

7.4 Temporisation du canal de communication point à point unidirectionnel direct

On distingue principalement deux types de comportement vis-à-vis de la propagation des signaux dans les circuits booléens [9]. Le mode « transport » propage n'importe quelle impulsion à travers une porte. Le mode « inertie » filtre les impulsions de largeur inférieure au temps d'activation de la porte en question.

Dans le cas « transport », on produit simplement une notification retardée (voir III3.4.6). Ce mode suffit à notre étude. Le cas de la propagation inertielle des impulsions se rapproche déjà d'un comportement « analogique » plus fin.

On complète le modèle du canal de communication point à point unidirectionnel direct par des variables de délais, i.e. un délai pour une transition montante et un délai pour une transition descendante.

On introduit donc dans le protocole de communication un troisième niveau de simulation correspondant au niveau temporisé, le niveau 2 dans l'interface d'entrée et dans le canal.

Pseudo code de l'interface d'entrée du canal :

```
1 interface_entrée_canal_sortie_port {
2     // méthodes
3     produit_requête(entier niveau_temporisation, délai
délai_requête) ;
4     consomme_acquittement() ;
5     écrit_valeur(booléen valeur) ;
6     // variables
7     événement requête ;
8     booléen valeur_entrée ;
9 }
```

Pseudo code de la méthode de production de requête du canal :

```
1 produit_requête(entier niveau_temporisation, délai délai_requête) {
2     si niveau_temporisation = 0
```

```
3           requête.notifie() ;
4     si niveau_temporisation = 1
5           requête.notifie(délai_nul) ;
6     si niveau_temporisation = 2
7           requête.notifie(délai_requête) ;
8  }
```

7.5 Temporisation du canal de communication point à point unidirectionnel à poignée de main

Le producteur se synchronise avec le reste du circuit grâce au protocole de communication point à point unidirectionnel à poignée de main que nous avons présenté au paragraphe 3.3.

Symétriquement, on enrichit le protocole de communication avec une variable de niveau de temporisation pour distinguer entre les différents types de notifications pour l'acquittement. Voici le pseudo code correspondant.

Pseudo code de l'interface de sortie du canal :

```
1  interface_sortie_canal_entrée_port {
2     // méthodes
3     produit_acquittement(   entier niveau_temporisation,
4                             délai délai_acquittement) ;
5     consomme_requête() ;
6     met_à_jour_valeur_canal() ;
7     booléen lit_valeur() ;
8     // variables
9     événement acquittement ;
10    booléen valeur_sortie;
11 }
```

Pseudo code de la méthode de production de requête du canal :

```
1  produit_acquittement(entier niveau_temporisation,
2                       délai délai_acquittement) {
3     si niveau_temporisation = 0
4           acquittement.notifie() ;
5     si niveau_temporisation = 1
6           acquittement.notifie(délai_nul) ;
7     si niveau_temporisation = 2
8           acquittement.notifie(délai_acquittement) ;
9     }
```

Par conséquent, on obtient 3 possibilités de temporisation pour la requête et 3 possibilités de temporisation pour l'acquittement. Cela nous donne 9 configurations de temporisation en tout pour le protocole de communication point à point unidirectionnel à poignée de main. Chacune de ces configurations correspond à une temporisation spécifique du canal de communication.

Niveau	Requête	Acquittement
0	0	0
1	0	1
2	0	2
3	1	0
4	1	1
5	1	2
6	2	0
7	2	1
8	2	2

Figure 35 : Configurations de temporisation du protocole de communication point à point unidirectionnel à poignée de main

Examinons ces différentes configurations lorsque le tirage aléatoire des processus est activé dans le noyau de simulation.

Ceux où apparaît une notification immédiate peuvent causer une rupture de la relation de causalité à l'initialisation. Il s'agit des niveaux 0, 1, 2, 3 et 6.

Tous les niveaux où la requête intervient après un délai non nul correspondent à un retard à l'initialisation. Or l'initialisation a lieu dès le premier pas de calcul. Nous excluons donc les niveaux 7 et 8.

Il reste les niveaux 4 et 5 pour le protocole de communication point à point unidirectionnel à poignée de main. Ces configurations correspondent respectivement aux niveaux 1 et 2 du protocole de communication point à point unidirectionnel direct.

Ces configurations de temporisation restent valables lorsque le tirage aléatoire des processus n'est pas activé dans le noyau de simulation.

7.6 Résumé

Dans notre modèle, la temporisation est prise en compte au niveau des canaux de communication. Nous choisissons d'implanter le mode « transport » de propagation des impulsions.

Conformément à l'algorithme d'ordonnancement de SystemC v2.0.1, nous obtenons trois niveaux de temporisation pour un canal de communication point à point unidirectionnel direct :

- Niveau 0 : requête immédiate, sensibilisation dynamique des processus
- Niveau 1 : requête à délai nul,
- Niveau 2 : requête à délai non nul.

Dans le cas du producteur, nous obtenons trois niveaux de temporisation pour un canal point à point unidirectionnel à poignée de main :

- Niveau 0 : requête immédiate, acquittement immédiat, sensibilisation dynamique des processus

- Niveau 1 : requête à délai nul, acquittement à délai nul
- Niveau 2 : requête à délai nul, acquittement à délai non nul.

Dans le cas temporisé, on peut spécifier un délai pour une transition montante et un délai pour une transition descendante dans le canal de communication point à point unidirectionnel. Dans le cas du protocole direct, le délai est pris en compte au moment de la production d'une requête. Dans le cas du protocole à poignée de main, le délai est pris en compte au moment de la production d'un acquittement.

On ne conserve le niveau 0 que par souci de complétude. En fait, dans le cas des systèmes à événements discrets asynchrones, il est réservé à la phase d'élaboration des liens entre événements et processus. La temporisation est cohérente avec le type d'ordonnancement du niveau 1 et du niveau 2, i.e. sans perte de comportement significatif dans le passage d'un niveau à l'autre.

Dans la suite, on qualifie de « non temporisé », le comportement de niveau 1 et de « temporisé » le comportement de niveau 2.

8 Mise en œuvre : l'oscillateur de Muller

8.1 Spécification

Considérons la spécification sous forme de réseau de Petri ordinaire de la Figure 36. Il s'agit d'un oscillateur dont la trace d'exécution est $(T_0T_1)^*$. On appelle traditionellement cet oscillateur, l'oscillateur de Muller [11], qui vérifie les propriétés d'insensibilité aux délais développées au paragraphe I6.2.

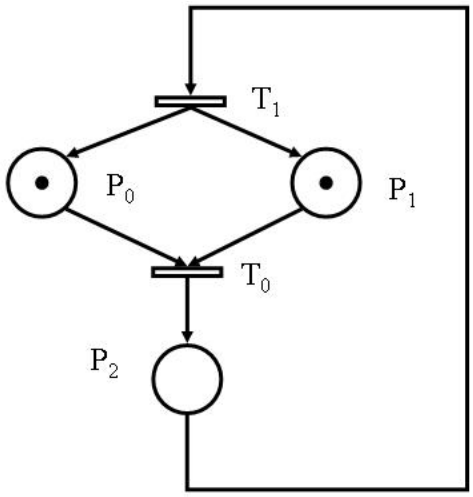


Figure 36 : Réseau de Petri ordinaire de l'oscillateur de Muller

8.2 Traduction de Patil

On peut se baser sur la méthode de Patil présentée au chapitre II pour traduire cette spécification sous forme de circuit insensible aux délais. On obtient alors le circuit de la Figure 37.

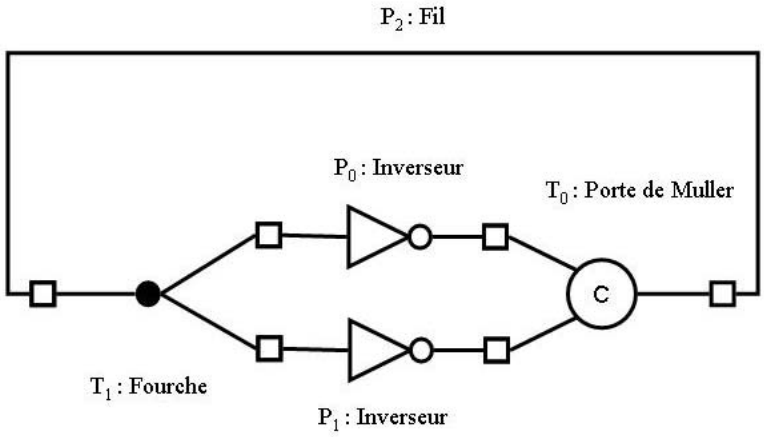


Figure 37 : Circuit de Patil de l'oscillateur de Muller

8.3 Modélisation non temporisée

Si l'on se contente d'implémenter directement ce circuit avec les éléments présentés au paragraphe 4, le circuit ne peut pas fonctionner correctement. En effet, tous les processus se synchronisent avec le simulateur à l'initialisation et aucun d'eux ne produit la première notification nécessaire au démarrage de la simulation.

Il faut donc introduire les éléments nécessaires à l'initialisation du circuit comme on l'a évoqué au paragraphe 6. En termes de réseau de Petri, cela revient à produire le marquage initial du réseau. On modifie donc le jeu de traduction de Patil. Plusieurs solutions sont possibles. Dans l'esprit de la traduction de Patil, c'est l'inverseur qui traduit une place marquée. Nous choisissons par conséquent de transformer les inverseurs en portes « non-ou » et d'introduire un producteur effectuant la mise à zéro. Ce producteur est ensuite connecté aux portes « non-ou » grâce à une fourche. La Figure 38 nous montre la nouvelle règle de transformation et le circuit obtenu.

Le producteur émet la valeur « 1 » à l'initialisation. Cette valeur se propage à travers la fourche jusqu'aux portes « non-ou ». Cela a pour effet de mettre à « 0 » la sortie des « non-ou » et donc d'éliminer le marquage initial. Le producteur émet ensuite un « 0 », introduisant ainsi le marquage initial sur la sortie des « non-ou » et provoquant le début de l'oscillation.

Ce circuit n'est plus insensible aux délais. Pour mettre en œuvre la spécification, il a fallu traiter le marquage initial et donc « ouvrir » le circuit. Nous sortons alors du cadre du théorème sur l'abstraction des délais du paragraphe I6.2. En effet, la spécification n'est plus « fermée ».

Plus précisément, on a été conduit à introduire une fourche, nommée « FRAZ » sur le schéma de la Figure 38. Si le producteur évolue trop vite, il risque de ne pas respecter la propriété de persistance de la fourche. On met ce problème en évidence au chapitre V.

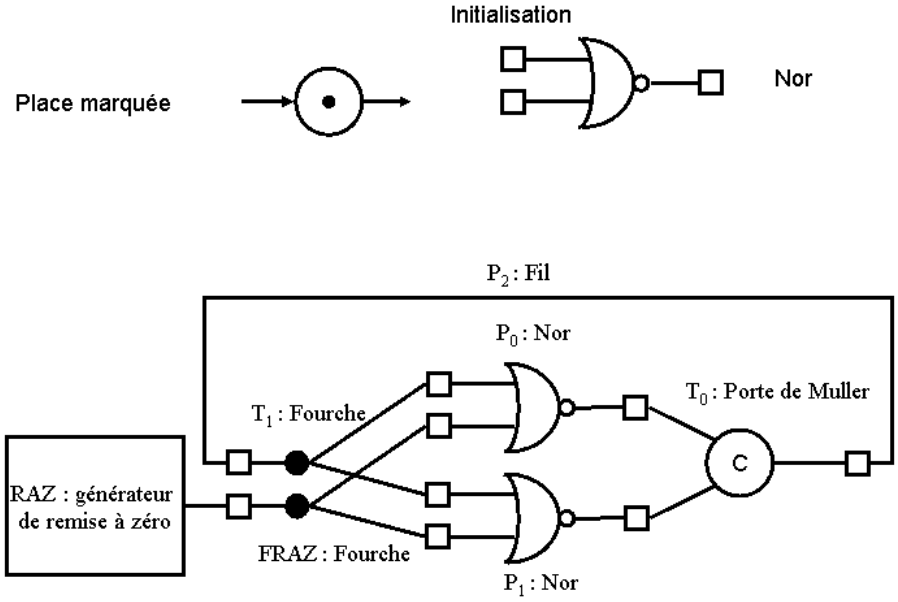


Figure 38 : Circuit de Patil de l'oscillateur de Muller avec mise à zéro

On trouve le pseudo code du modèle du circuit de Patil de l'oscillateur de Muller avec mise à zéro à l'annexe C.11.

8.4 Simulation non temporisée

8.4.1 Respect de la relation de causalité à l'initialisation

Considérons le producteur. Sur le premier pas de calcul, l'adaptateur entre dans sa phase d'initialisation et se synchronise sur son entrée. Il attend un événement de la part du générateur.

Sur le même pas de calcul, le générateur déclenche un événement à délai nul, puis se synchronise sur sa sortie. Il attend un événement de son environnement pour produire un nouveau stimulus.

L'événement déclenché par le générateur est nécessairement traité après l'initialisation de tous les processus. En effet, cette initialisation a lieu dans la boucle de niveau 0 alors que le premier stimulus est traité au pas de calcul suivant dans la boucle de niveau 1.

La relation de causalité à l'initialisation est respectée.

8.4.2 Simulation avec ou sans tirage aléatoire

Par conséquent, que ce soit avec ou sans tirage aléatoire, cette spécification étant insensible aux délais (mis à part la mise à zéro). La simulation est correcte. Dans le cas où le tirage aléatoire est activé, on observe différents entrelacements entre des processus parallèles au cours de la simulation. Ce qui n'est pas le cas lorsque le tirage aléatoire n'est pas activé.

On donne un chronogramme de la simulation non temporisée sur la Figure 39. SystemC v2.0.1 fournit un mode de génération de traces au format VCD (« Value Change Dump » en anglais). SystemC v2.0.1 fournit ce mode aussi bien au niveau 1, qu'au niveau 2. Il existe des outils permettant de visualiser l'évolution du circuit en interprétant ce format de traces. Ici,

chaque pas de délai nul est interprété comme un pas de délai minimal dans la résolution choisie pour la simulation temporisée.

On observe :

- au pas 0 l'initialisation des processus,
- au pas 1 l'écriture d'un « 1 » à travers le canal raz_fraz,
- au pas 2 la recopie du « 1 » sur les canaux fraz_p0 et fraz_p1,
- au pas 3 l'écriture d'un « 0 » sur le canal raz_fraz,
- au pas 4 la recopie sur les canaux fraz_p0 et fraz_p1.

La mise à zéro est terminée. Ensuite on observe que :

- au pas 5, les canaux p0_t0 et p0 t1 écrivent un « 1 » à l'entrée de la porte de Muller,
- au pas 6 la porte de Muller bascule,
- au pas 7, le fil recopie un « 1 » sur les canaux t1_p0 et t1_p1,
- au pas 8, les entrées des portes « non_ou » basculent,
- au pas 9, les entrées de la porte de Muller basculent,
- au pas 10, la porte de Muller bascule,
- au pas 11, le basculement de la porte de Muller s'est propagé jusqu'à l'entrée de la fourche T_1.

Et ainsi de suite. On a atteint un point fixe.

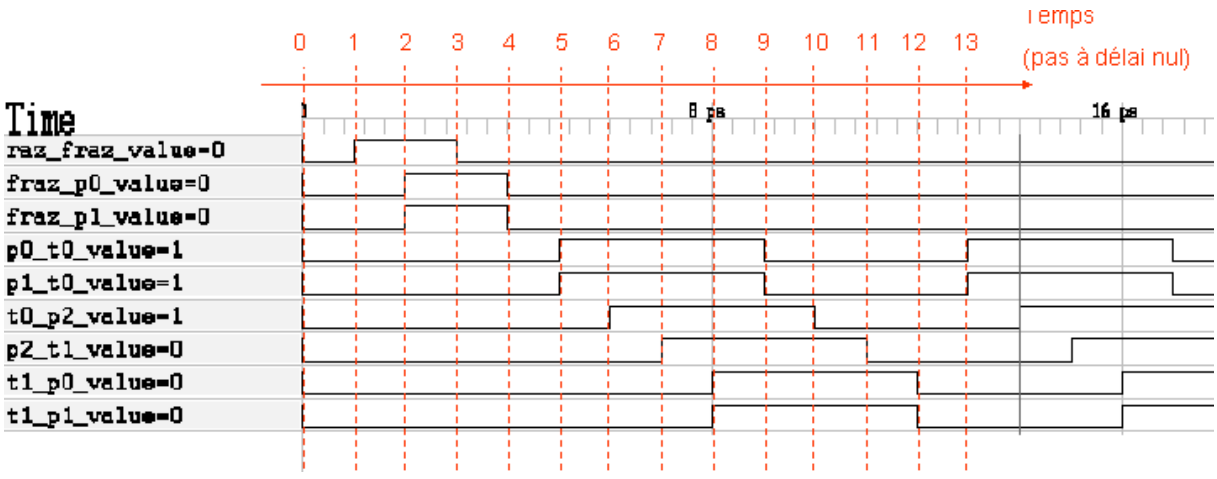


Figure 39 : Chronogramme de simulation non temporisée de niveau 1 de l'oscillateur de Muller

8.5 Modélisation temporisée

Sur le schéma de la Figure 40, chaque petit carré représente un canal de communication point à point unidirectionnel direct. Le petit hexagone entre le générateur et l'adaptateur représente un canal de communication point à point unidirectionnel à poignée de main. Pour simplifier, nous choisissons de temporiser ces éléments avec la même valeur pour le délai de la transition montante et le délai de la transition descendante.

Le producteur (porte « RAZ ») et la fourche qui le suit (porte « FRAZ ») sont des éléments sensibles aux délais. On choisit un délai suffisamment long entre le générateur et l'adaptateur pour laisser le temps aux signaux de s'établir convenablement dans les deux branches de la fourche qui suit. Ainsi la propriété de persistance de cet élément sera respectée.

Nous étudierons ce comportement plus en détail au chapitre V.

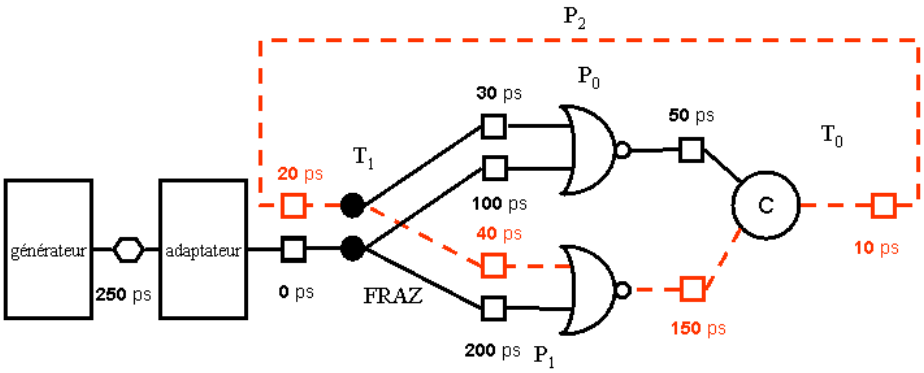


Figure 40 : Circuit de Patil de l'oscillateur de Muller avec mise à zéro et temporisation

Nous ajoutons au modèle du circuit de l'oscillateur de Muller les délais dans les canaux. Le délai passé au producteur représente le délai du canal de communication point à point unidirectionnel à poignée de main qui relie le générateur et l'adaptateur (voir annexe B.3).

8.6 Simulation temporisée

La simulation de niveau 2 obtenue est correcte. Le diagramme de simulation obtenu est reporté sur la Figure 41.

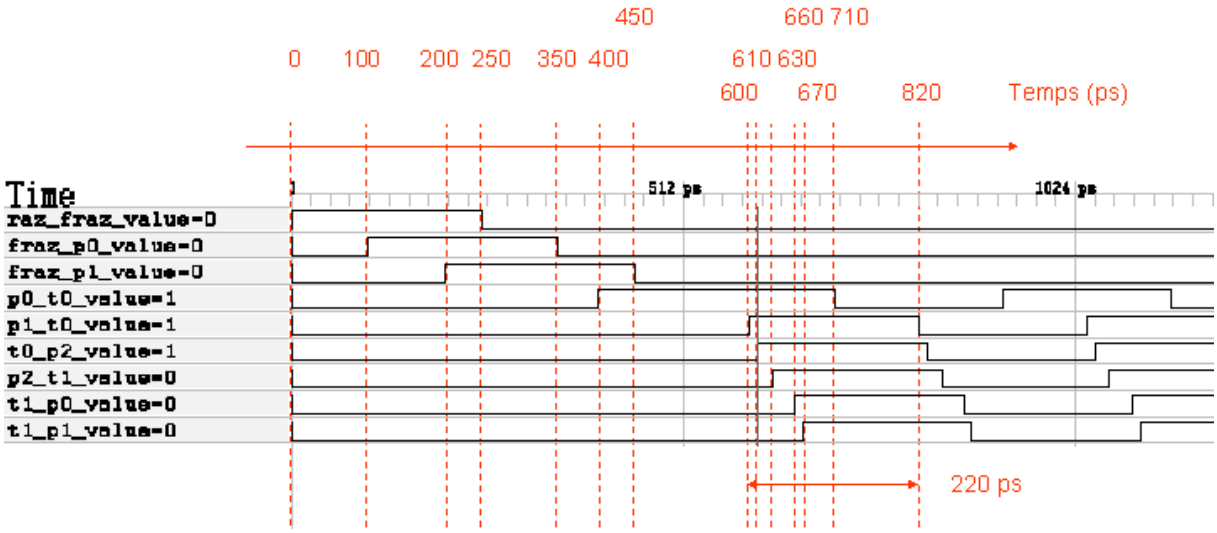


Figure 41 : Chronogramme de simulation temporisée de l'oscillateur de Muller

Au niveau de la génération de stimuli, les signaux « raz_fraz », « fraz_p0 » et « fraz_p1 » montrent que les valeurs ont le temps de s'établir à la sortie de la fourche.

Il faut 250 ps + 0 ps + 200 ps + 150 ps = 600 ps au circuit pour s'initialiser. Une fois la phase de mise à zéro terminée, le circuit se met à osciller. La demi-période d'oscillation (10 ps + 20 ps + 40 ps + 150 ps = 220 ps) obtenue correspond au chemin le plus long dans le circuit. C'est une conséquence de l'insensibilité aux délais du circuit. En fait, non seulement s'agit-il du chemin le plus long dans le circuit, mais cette longueur n'est pas nécessairement bornée. Ce chemin est représenté en pointillés sur la Figure 40.

9 Résultats

Finalement, nous obtenons un modèle de circuits numériques asynchrones, s'intégrant dans le standard SystemC v2.0.1 de conception de systèmes numériques. Ce modèle consiste en 4 niveaux comme le montre la Figure 42.

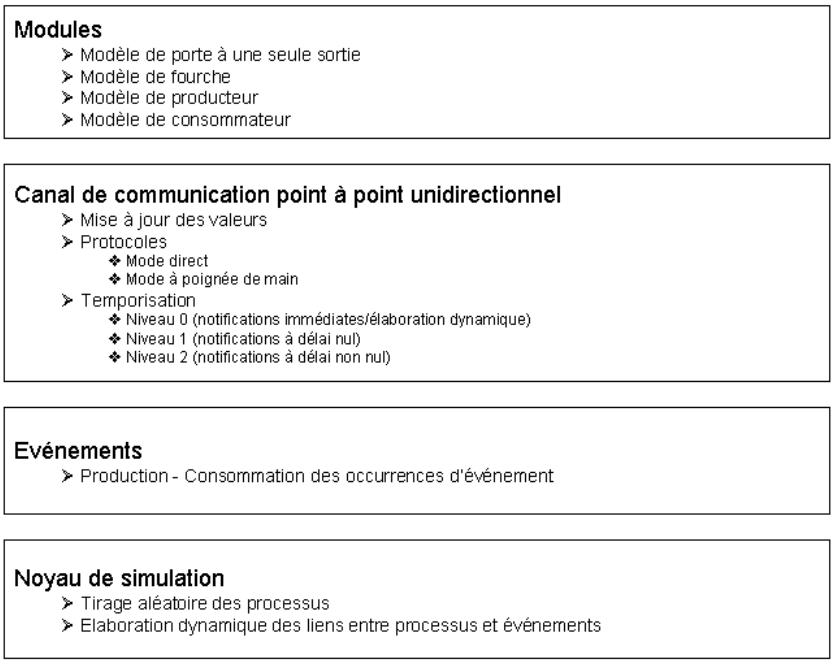


Figure 42 : Modèle de circuits numériques asynchrones, s'intégrant dans le standard SystemC v2.0.1 de conception de systèmes numériques

Le premier niveau concerne la couche « noyau de simulation » de SystemC v2.0.1. La modification permet de supporter la sémantique des systèmes à événements discrets asynchrones en intégrant le tirage aléatoire des processus dans le noyau. Dans ce contexte, on a interdit le recours à l'ordonnancement de niveau 0. La sensibilisation dynamique des processus est associée au niveau 0 de l'algorithme de simulation.

Le second niveau concerne la couche « événements » de SystemC v2.0.1. La modification consiste à introduire un couplage minimal entre noyau de simulation et modèle SystemC v2.0.1 pour augmenter la puissance d'expression de l'algèbre de synchronisation du langage. Un modèle SystemC v2.0.1 et le noyau de simulation communiquent alors suivant un protocole de production-consommation des occurrences d'événements. La conséquence est la possibilité d'implémenter des structures de choix et ainsi d'élever la puissance d'expression de l'algèbre sur événement de SystemC v2.0.1 au niveau des réseaux de Petri ordinaires.

Le troisième niveau concerne la couche « communication » de SystemC v2.0.1. La modification consiste à introduire un nouveau canal de communication hiérarchique point à point unidirectionnel. Ce canal supporte la mise à jour des valeurs suivant le protocole de production-consommation des occurrences d'événements introduit au niveau « événements ». Il supporte deux protocoles de communication : le mode direct et le mode à poignée de main. Il supporte également 3 niveaux de temporisation : le niveau 0 ou immédiat, le niveau 1 ou à délai nul et le niveau 2 ou à délai non nul. Bien que réservé à la sensibilisation dynamique des processus, le niveau 0 peut être activé au niveau du canal.

Le quatrième niveau concerne la couche « modèles de calcul » de SystemC v2.0.1. La modification consiste à introduire les modèles spécifiques aux circuits numériques asynchrones à partir des circuits quasi insensibles aux délais. On trouve un modèle de porte à une seule sortie et un modèle de fourche. Pour résoudre le problème de l'initialisation de la simulation dans le cas d'un circuit numérique asynchrone et pour compléter le modèle de calcul, on introduit un producteur et un consommateur. La porte à une seule sortie, la fourche et le consommateur s'appuient sur le protocole de communication point à point direct. Le producteur se divise en deux modèles : un générateur et un adaptateur. Ces deux modèles sont reliés par un canal de communication point à point unidirectionnel à poignée de main. L'adaptateur réalise la conversion entre le mode à poignée de main et le mode direct.

Nous mettons ces résultats en œuvre au paragraphe 8 sur un exemple de conception de circuit insensible aux délais : l'oscillateur de Muller. Nous pouvons modéliser ce circuit correctement à condition de prendre compte la phase de mise à zéro.

Nous pouvons simuler un tel circuit sans activer le tirage aléatoire des processus. Cela correspond à la sémantique actuelle de SystemC v2.0.1. Nous pouvons aussi simuler un tel circuit en activant le tirage aléatoire implanté dans le noyau de simulation pour respecter la sémantique d'exécution d'un système à événements discrets asynchrone.

Nous pouvons également faire varier le type de simulation en jouant sur le niveau de temporisation :

- niveau 0 : notifications immédiates, sensibilisation dynamique des processus,
- niveau 1 : notifications à délai nul, simulation non temporisée,
- niveau 2 : notifications à délai non nul, simulation temporisée.

Nous venons d'atteindre notre objectif, puisque nous sommes maintenant capable de supporter la sémantique des systèmes à événements discrets asynchrones dans SystemC v2.0.1. Nous sommes capables de modéliser et de simuler dans un cadre standard des circuits numériques asynchrones. La puissance d'expression de l'algèbre de synchronisation des processus de SystemC v2.0.1 a été élevée au niveau des réseaux de Petri ordinaires. Nous pouvons activer le tirage aléatoire des processus dans le noyau et temporiser le modèle de façon cohérente en cohérence.

Ainsi nous supportons les quatre premières phases du flot descendant de conception des circuits numériques asynchrones que nous adressons dans l'environnement standard SystemC v2.0.1 :

- modélisation non temporisée,
- simulation non temporisée,
- temporisation du modèle,
- simulation temporisée.

Nous voyons au chapitre V comment enrichir ce modèle pour prendre en compte la vérification des propriétés d'insensibilité aux délais d'un circuit numérique asynchrone, i.e. persistance, sûreté et vivacité.

V Vérification de circuits numériques asynchrones dans SystemC v2.0.1

L'objectif de ce chapitre de contribution est multiple. Nous formalisons en LTL les propriétés d'insensibilité aux délais, i.e. persistance, sûreté et vivacité des circuits numériques asynchrones quasi insensibles aux délais. A partir de là, nous instrumentons le modèle de circuits numériques asynchrones pour la vérification des propriétés d'insensibilité aux délais. Puis, nous étudions la composition de cette instrumentation avec la temporisation du modèle. Enfin, nous mettons en œuvre le modèle obtenu sur plusieurs circuits de taille réduite, mais significatifs. La contribution de ce chapitre est un modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais, s'intégrant dans le standard SystemC v2.0.1 de conception de systèmes numériques.

1 Introduction

1.1 Problématique de la vérification

La problématique de la vérification d'un système à événements discrets se sépare en deux branches : la vérification statique et la vérification dynamique.

La vérification *statique* s'appuie sur la sémantique statique du langage de spécification. Cette sémantique statique est définie par le lexique, la grammaire non contextuelle et éventuellement la grammaire contextuelle du langage de spécification.

La vérification *dynamique* s'appuie sur la sémantique dynamique du langage de spécification. Cette sémantique dynamique est définie suivant une méthode formelle. Il existe plusieurs types de formalisations. La plus abstraite, appelée notationnelle consiste en une description mathématique fonctionnelle. La plus concrète, appellée opérationnelle consiste en une description sous forme de système de transitions étiquetées.

En informatique, la sémantique dynamique est le plus souvent opérationnelle. C'est le niveau de détail nécessaire à la spécification du compilateur d'un langage. On crée un modèle correspondant à la spécification en associant des valeurs aux variables de la sémantique opérationnelle.

Dans le contexte de la vérification, le travail du théoricien consiste à étudier la décidabilité et la complexité des problèmes à traiter. On s'assure dans un premier temps que l'on peut répondre aux questions que pose la vérification. Une fois que l'on a assuré la décidabilité des problèmes à traiter, on quantifie le temps et l'espace mémoire qui sont nécessaires au traitement des problèmes soulevés.

Dans le cas d'un système à événements discrets asynchrone décrit dans le cadre des réseaux de Petri ordinaires, on sait que la vérification des propriétés de persistance, sûreté et vivacité est un problème décidable et en général PSPACE. On se reportera au paragraphe I5.6 pour plus de détails sur le sujet.

Les méthodes formelles, dérivant de la logique mathématique, forment un champ à part entière (enseignement, recherche, industrie) dont une présentation exhaustive sort du cadre de cette thèse. Il s'agit ici de dégager les axes principaux dans le but de positionner notre

approche et de faire l'état de l'art vis-à-vis de SystemC v2.0.1. Pour en savoir plus sur les méthodes formelles, on pourra se tourner vers la référence [71] qui représente un bon pointeur vers des ouvrages de synthèse, des publications, des outils, des supports de cours, etc.

1.2 Vérification statique

On s'intéresse ici à la structure du programme décrivant le système et à sa conformité vis-à-vis de « bonnes règles » de conception. Cette phase de vérification est implantée dans la partie frontale du compilateur du langage. Il s'agit d'éliminer statiquement quand c'est possible le code mort, des structures de programme indésirables, de signaler des fonctions trop longues, du code non documenté en plus de la vérification des types et des indices. Cela permet de détecter des erreurs de programmation, d'améliorer les performances et de faciliter la maintenance de la spécification.

1.3 Vérification dynamique

On s'intéresse ici au comportement du système. On peut classer les techniques de vérification dynamique suivant trois critères : le type de théorie, le type de simulation et le type de vérification.

Il existe principalement deux supports théoriques pour la vérification dynamique d'une spécification :

- la théorie des modèles (« model checking » en anglais),
- la théorie de la démonstration.

Dans le premier cas, on évalue le modèle sur divers domaines de valeurs. Dans le second, on démontre des théorèmes dans le cadre de la spécification.

Il existe deux types de simulation :

- les simulations exhaustives,
- les simulations non exhaustives.

L'approche exhaustive consiste à considérer tous les comportements possibles du modèle. L'approche non exhaustive se limite à la simulation du modèle dans certaines configurations.

Il existe également deux types de vérification :

- la preuve d'équivalence,
- la vérification de propriétés.

Dans le premier cas, il s'agit de montrer que le modèle est équivalent à un modèle de référence. Dans le deuxième cas, il s'agit de montrer que le modèle satisfait certaines propriétés.

1.4 Vérification formelle dans SystemC v2.0.1

1.4.1 Formalisation

Une équipe de chercheurs de l'université de Tübingen s'est penchée en 2001 sur la formalisation du langage SystemC et du noyau de simulation à l'aide d'ASM (« distributed Abstract State Machines » en anglais) [63]. Le choix de cette formalisation a été guidé par la volonté de rendre SystemC interopérable avec VHDL et Verilog dont la sémantique a été

formalisée dans le même cadre. Ce développement concerne la version 1.0 de SystemC et nécessite une mise à jour pour la version 2.0.1.

1.4.2 Approche exhaustive

Des chercheurs ont étudié en 2002 la mise en oeuvre de techniques de simulation exhaustive pour des circuits synchrones décrits au niveau transfert de registre dans l'environnement SystemC v2.0.1 [64].

Leur objectif était de construire le graphe d'accessibilité du circuit à partir de sa description dans SystemC v2.0.1. Pour cela, ils avaient besoin d'une représentation sous forme de BDD (« Binary Decision Diagram » en anglais) pour chaque entrée de bascule du circuit et de la fonction de calcul de l'état suivant. Ils ont choisi de récupérer la structure du circuit à la fin de la phase d'élaboration de SystemC v2.0.1 par la prise prévue à cet effet (méthode « end_of_elaboration ») et de développer leur propre simulateur exhaustif.

Ils ont donc introduit une classe contenant une table de hachage pour stocker la structure du circuit et les méthodes nécessaires à la construction et à l'évaluation des BDD, au calcul de l'état suivant et enfin à la construction du graphe d'accessibilité.

Cette méthode a été appliquée à un arbitre redimensionnable à priorité fixe et tourniquet pour un bus embarqué. Les résultats montrent bien la variation exponentielle de la place mémoire et du temps de calcul en fonction de la taille du circuit.

1.4.3 Approche non exhaustive

Des travaux ont été publiés en 2000 sur le rapprochement des techniques de simulation non exhaustive et de vérification de propriétés en LTL pour la validation des systèmes logiciels/matériels synchrones.

Une technique de vérification de propriétés en LTL basée sur l'environnement de spécification et de simulation SystemC v1.0 [65] a été développée. SystemC v1.0 a été choisi non seulement à cause de sa vitesse d'exécution mais aussi pour les facilités de modélisation offertes à différents niveaux d'abstraction.

Leur technique permet de spécifier des propriétés en LTL en étendant le mécanisme d'assertion présent dans le simulateur. Le choix est donc laissé au concepteur d'insérer dans la spécification des points de vérification où cela lui semble nécessaire. Des machines à états chargées de la vérification de ces propriétés sont générées à la volée et évoluent en parallèle de la simulation.

Une logique à trois états (vrai, faux, pendant (« pending » en anglais)) est introduite pour prendre en compte la finitude des traces de simulation comparativement à une approche symbolique qui permettrait de raisonner sur un jeu de traces infini.

Cette méthode a été appliquée à un arbitre redimensionnable à priorité fixe et tourniquet pour un bus embarqué. Le surcoût obtenu en terme de temps de simulation dans ce cas est de l'ordre de 100%. Le surcoût en terme d'utilisation mémoire n'est pas indiqué.

1.5 Contribution

Chaque méthode de vérification présentée brièvement dans l'introduction possède ses avantages et ses inconvénients. Toutes ces approches sont complémentaires.

La vérification statique requiert un bon compilateur. La démonstration de théorèmes est souvent semi automatique. L'approche exhaustive est limitative en terme de taille de circuit. La preuve d'équivalence est conditionnée par l'existence d'une référence.

Dans le contexte de SystemC v2.0.1, on dispose d'un compilateur standard C++ pour la vérification statique. Sur le plan de la vérification dynamique, on se trouve naturellement aiguillé vers une approche non exhaustive. Cette approche permet de gérer des modèles de grande complexité mais reste une sorte de preuve par 9. En effet, lorsque l'on ne trouve plus d'erreur, rien ne garantit que le modèle respecte parfaitement la spécification. Les méthodes exhaustives entrent en jeu à ce moment-là du flot descendant de conception des systèmes à événements discrets.

Les propriétés d'insensibilité aux délais d'un système à événements discrets nous permettent d'inscrire la conception d'un tel système dans un flot descendant. En effet, en l'absence de référence, nous savons que notre système est correct s'il satisfait les propriétés de persistance, de sûreté et de vivacité. On complète notre modèle de circuits numériques asynchrones avec des instruments permettant de vérifier persistance, sûreté et vivacité au cours d'une simulation non exhaustive. C'est la contribution de ce chapitre.

Nous voyons que cette instrumentation a peu d'impact sur la sémantique initiale du modèle de circuits numériques asynchrones développée au chapitre IV et se compose de manière cohérente avec ce dernier.

Enfin, nous illustrons ces concepts au paragraphe 5 à travers l'exemple présenté au chapitre IV : l'oscillateur de Muller. Cet exemple nous permet de valider l'instrumentation de la vérification des propriétés de persistance et de vivacité. Nous voyons que la rupture de la causalité à l'initialisation déclenche un blocage, que la temporisation de la circuiterie de mise à zéro peut violer la persistance.

Pour la validation de l'instrumentation de la vérification de la propriété de sûreté, nous développons un exemple ad hoc : le générateur de jetons.

Notre flot de conception concerne alors les 4 phases suivantes du flot de conception descendant :

- modélisation non temporisée,
- simulation non temporisée *et vérification des propriétés d'insensibilité aux délais*,
- temporisation du modèle,
- simulation temporisée *et vérification des propriétés d'insensibilité aux délais*.

2 Formalisation des propriétés d'insensibilité aux délais

2.1 Persistance

On a défini la persistance dans le cadre des réseaux de Petri ordinaires au paragraphe I5.3. Il s'agit de vérifier l'atomicité d'une action dans un système à événements discrets.

Comme on l'a vu au paragraphe IV4.1, en LTL, l'opérateur U permet de spécifier une commande gardée assortie de la contrainte de persistance de la garde :

- $(\sigma, j) \models pUq \Leftrightarrow$ il existe $k \geq j, (\sigma, k) \models q$ et pour tout i tel que $j \leq i < k, (\sigma, i) \models p$.

C'est la portion « pour tout i tel que $j \leq i < k, (\sigma, i) \models p$ » garantissant que la garde est vraie de l'instant d'indice j à l'instant d'indice $k-1$ qui représente cette contrainte de persistance.

On a choisi aux paragraphes IV4.2 et IV4.3 de formaliser une règle de production à une ou plusieurs sorties par l'intermédiaire de cet opérateur U .

La contrainte de persistance était donc déjà prise en compte dans la formalisation des règles de production aux paragraphes IV4.2 et IV4.3.

2.2 Sûreté

On a défini la sûreté dans le cadre des réseaux de Petri ordinaires au paragraphe I5.4. Cette violation de la propriété de sûreté peut être le résultat de la violation d'une propriété d'exclusion mutuelle dans une structure de choix quelque part dans le réseau. Il s'agit alors de vérifier l'exclusion mutuelle entre les gardes des règles de production d'une porte tout au long de la simulation.

Comme on l'a vu au paragraphe IV4.1, en LTL l'opérateur $\Box$ signifiant « pour toujours » permet capturer cette notion de « tout au long de la simulation ».

On a vu a paragraphe IV4.4 qu'une porte à une seule sortie était définie par :

- $\begin{cases} p_0 \to q \uparrow \Leftrightarrow (\sigma, j) \models p_0 U q \\ p_1 \to q \downarrow \Leftrightarrow (\sigma, j) \models p_1 U \neg q \end{cases}$

Il s'agit donc de vérifier la propriété suivante :

- $(\sigma, 0) \models \Box\left(\neg p_0 \vee \neg p_1\right)$

Il en va de même pour la fourche, cette propriété étant indépendante du nombre de sorties de la porte. Cette propriété doit être vérifiée pour toutes les portes du circuit dès l'instant initial.

2.3 Vivacité

On a défini la vivacité dans le cadre des réseaux de Petri ordinaires au paragraphe I5.5. Il s'agit de vérifier qu'il existe toujours une transition tirable dans le réseau pendant la simulation.

Comme on l'a vu au paragraphe IV4.1, en LTL l'opérateur $\Diamond$ signifiant « inéluctablement » permet de capturer la notion d'occurrence. Au paragraphe IV4.2, on a vu qu'une transition était déclenchée par une garde dans une règle de production. Il s'agit donc de vérifier qu'à tout instant dans la simulation, il exite au moins une garde qui va devenir vraie. Notons alors G l'ensemble des gardes du circuit. La contrainte suivante doit être respectée :

- pour tout $j \geq 0$, il existe $p \in G$ tel que $(\sigma, j) \models \Diamond p$.

3 Vérification des propriétés d'insensibilité aux délais

3.1 Persistance

On a profité de la capacité de SystemC v2.0.1 de séparer calcul et communication (voir paragraphe IV3). On a encapsulé la communication entre les portes dans un canal de communication point à point unidirectionnel. D'une part, ce canal est chargé de véhiculer les événements entre les processus codant les règles de production. D'autre part, il encapsule des variables partagées entre les deux règles de production portant les valeurs à l'entrée et à la sortie du canal.

On introduit une nouvelle variable partagée dans le canal de communication point à point unidirectionnel indiquant si une communication est en cours. Cette variable booléenne est nommée « verrou ».

Elle correspond à un sémaphore à variable booléenne et représente la portion suivante de la sémantique de l'opérateur U (« jusqu'à ») en LTL : « pour tout i tel que $j \leq i < k, (\sigma, i) \models p$ ». C'est la proposition exprimant la persistance de l'action (voir paragraphe 2.1).

Nous modifions alors le protocole de communication entre les modules en conséquence.

Lors d'une écriture, lorsqu'un processus produit une occurrence d'événement sur un port de sortie, il commence par tester si le canal de sortie correspondant est déjà verrouillé ou pas. Si le canal est déjà verrouillé, cela revient à dire qu'une communication est en cours. L'atomicité de la règle de production et donc la propriété de persistance sont violées. Si le canal est déverrouillé, alors le processus verrouille ledit canal de sortie.

Lors d'une lecture, lorsque qu'un processus dans une porte est déclenché, il commence par tester si le canal d'entrée responsable du déclenchement est verrouillé ou pas. Dans une porte, c'est vrai si le processus en cours d'exécution est le premier sur les deux processus de la porte à consulter le canal. Ensuite, le processus déverrouille le canal d'entrée. Le deuxième processus de la porte trouve le canal d'entrée qui l'a déclenché déjà déverrouillé.

Nous reprenons le pseudo code développé au chapitre IV pour compléter l'interface du canal de communication point à point unidirectionnel. Les modifications (en gras) n'intervenant qu'au niveau du canal de communication point à point unidirectionnel, les modèles de la porte à une seule sortie, de la fourche, du producteur et du consommateur sont intacts.

Pseudo code de l'interface d'entrée du canal avec prise en compte de la persistance :

```
1  interface_entrée_canal_sortie_port {
2      // méthodes
3      verrouille_canal() ;
4      produit_requête(  entier niveau_temporisation,
5                        délai délai_requête) ;
6      consomme_acquittement() ;
7      écrit_valeur(booléen valeur) ;
8      // variables
9      événement requête ;
10     booléen valeur_entrée ;
11     booléen verrou ;
12 }
```

Pseudo code de l'interface de sortie du canal avec prise en compte de la persistance :

```
1  interface_sortie_canal_entrée_port {
2      // méthodes
3      produit_acquittement(   entier niveau_temporisation,
4                              délai délai_acquittement) ;
5      consomme_requête() ;
6      met_à_jour_valeur_canal() ;
```

```
7     déverrouille_canal() ;
8     booléen lit_valeur() ;
9     // variables
10    événement acquittement ;
11    booléen valeur_sortie;
12 }
```

Le pseudo code du canal de communication point à point unidirectionnel avec prise en compte de la persistance est reporté à l'annexe D.1.

3.2 Sûreté

Telle qu'énoncée au paragraphe 2.2, la propriété de sûreté n'est pas vérifiable par l'intermédiaire d'une simulation non exhaustive. On ne dispose que d'un nombre fini de pas de simulation. Si l'on ne peut pas vérifier qu'une propriété décidable est vraie, on vérifie que son contraire est faux. Auquel cas, on aura vérifié qu'une telle erreur ne survient pas pendant notre simulation non exhaustive.

Le contraire de la propriété $(\sigma,0) \models \Box(\neg p_0 \vee \neg p_1)$ s'exprime par :

- il existe $j \geq 0$ tel que $(\sigma, j) \models \Diamond(p_0 \wedge p_1)$.

Dans ce cas, il s'agit de déclencher une erreur de sûreté dès que les deux gardes d'une porte ou de la fourche deviennent vraies.

Pour implanter cette vérification dans notre modèle SystemC v2.0.1, on a recours à deux variables partagées entre les processus au niveau du module pour stocker la valeur des gardes. Ces variables sont donc visibles depuis les deux règles de production. On peut alors les utiliser pour détecter la condition $(\sigma, j) \models p_0 \wedge p_1$ et déclencher une erreur de sûreté à l'état d'indice j, dès qu'elle survient.

On fait l'hypothèse que les deux gardes sont fausses au début de l'exécution d'une porte. Il faut donc réinitialiser ces variables à la fin de l'exécution de la porte. On utilise le compteur de processus pour détecter la fin de l'exécution de ladite porte.

Le pseudo code du modèle SystemC v2.0.1 d'une porte mono entrée, mono sortie avec prise en compte de la sûreté est reporté à l'annexe D.2.

3.3 Vivacité

Telle qu'énoncée au paragraphe 2.3, la propriété de vivacité n'est pas vérifiable par l'intermédiaire d'une simulation non exhaustive. On ne dispose que d'un nombre fini de pas de simulation. Si l'on ne peut pas vérifier qu'une propriété décidable est vraie, on vérifie que son contraire est faux. Auquel cas on aura vérifié qu'une telle erreur ne survient pas pendant notre simulation non exhaustive.

Comme pour la propriété de sûreté, calculons donc la contraposée de la propriété de vivacité : pour tout $j \geq 0$, il existe $p \in G$ tel que $(\sigma, j) \models \Diamond p$.

On obtient : il n'existe pas $j \geq 0$, tel que pour tout $p \in G$, $(\sigma, j) \models \Box(\neg p)$.

Dans ce nouvel énoncé traduisant l'absence de blocage, on a encore besoin de tous les pas de simulation pour vérifier que les gardes du circuit ne restent pas toujours fausses à partir d'un tel instant $j \geq 0$, s'il existe.

Finalement, dans ce contexte, la formalisation en LTL ne nous fournit pas de support approprié pour dériver une instrumentation de la vérification de la propriété de vivacité.

Il faut changer de point de vue.

Nous considérons donc qu'on obtient un blocage lorsqu'on simule un programme pendant un certain temps que l'on nommera « infini » et qu'au bout de ce temps, le système n'a pas terminé son exécution dans un état acceptable. Cela pose trois problèmes. Comment représenter un temps infini à l'aide d'un simulateur disposant d'un nombre fini de pas de simulation ? Comment détecter que le système est bloqué au bout de ce temps infini ? Comment définir un état acceptable pour la fin de la simulation ?

Pour ce faire, on décide de sacrifier les deux derniers pas de simulation du modèle de temps du noyau de simulation. Ces deux instants deviennent deux instants spécifiques : l'instant « blocage », suivi de l'instant « infini ». Ces deux instants sont définis comme des pièges pour la simulation. Ils respectent les règles suivantes :

- tout instant borné est strictement inférieur à l'instant « blocage » et à l'instant « infini »,
- l'instant « blocage » est inférieur ou égal à lui-même,
- l'instant « blocage » est strictement inférieur à l'instant « infini »,
- l'instant « infini » est inférieur ou égal à lui-même.

Ainsi, si au bout d'un temps dit « infini », il reste à exécuter des occurrences d'événements à l'instant « blocage », i.e. des communications pendantes, le système est considéré comme bloqué. On détectera une telle occurrence d'événement par un test inséré dans le noyau de simulation comme le montre la Figure 43.

Pour utiliser cette nouvelle caractéristique du noyau de simulation, il faut pouvoir déclencher des occurrences d'événement à l'instant « blocage ». Pour cela, on utilise la fonctionnalité de minuterie (« timer » en anglais) de SystemC v2.0.1. La méthode de synchronisation d'un processus peut être invoquée avec un paramètre de temporisation et un événement (voir paragraphe III3.4.6). Une fois le délai passé en paramètre à la primitive de synchronisation écoulé, le processus mis en sommeil est systématiquement réveillé et la simulation se poursuit même si l'occurrence d'événement attendue ne s'est pas produite. Il suffit alors d'invoquer cette primitive avec un délai (en l'occurrence l'instant « blocage ») et de lancer la simulation pendant un temps infini pour détecter un blocage éventuel.

Lorsqu'une telle simulation s'arrête, si un blocage est détecté, c'est qu'il restait dans la pile du simulateur un processus synchronisé en attente d'une occurrence d'événement, occurrence qui n'est pas survenue avant l'instant nommé « infini ». C'est ce que nous définirons comme un état non acceptable à l'instant « infini ».

L'accès à cette fonctionnalité peut être activé depuis le programme principal. En pratique, cela configure d'une part les processus avec un appel aux primitives de synchronisation soit simples, soit avec minuterie. D'autre part, la simulation est configurée pour durer jusqu'à l'instant infini.

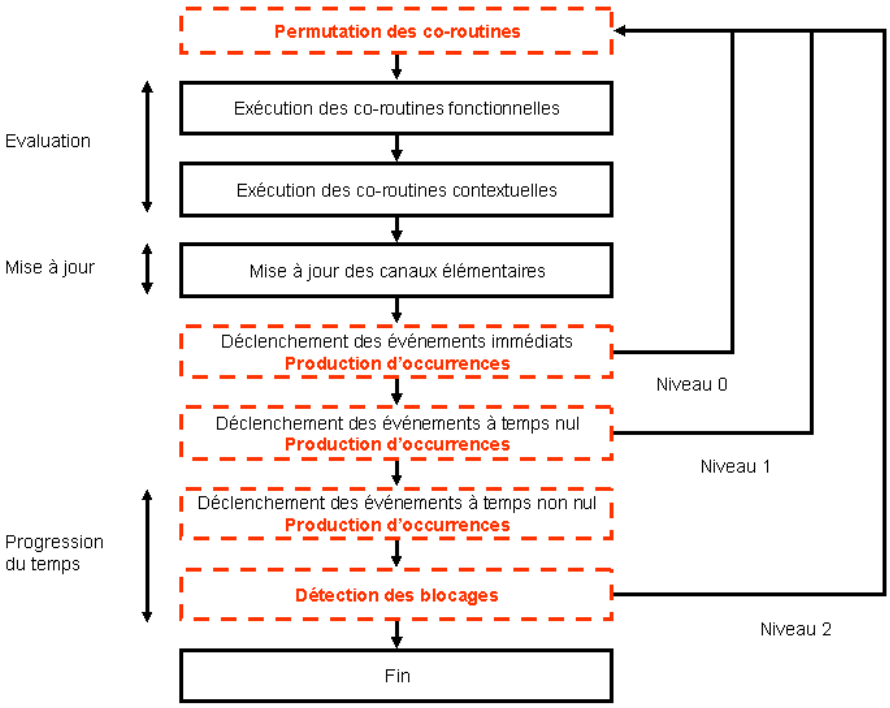


Figure 43 : Noyau de simulation agrémenté de la fonction de détection de blocage

Nous reprenons le pseudo code développé au chapitre IV pour décrire le générateur et l'adaptateur, sous-cellules du producteur. Ainsi, on verra comment écrire les primitives de synchronisation de ces modules aussi bien pour la requête que pour l'acquittement.

Le pseudo code du modèle SystemC v2.0.1 du producteur avec prise en compte de la vivacité est reporté à l'annexe D.3.2.

Le pseudo code du programme principal et avec activation de la détection des blocages est reporté à l'annexe D.3.4.

4 Vérification des propriétés d'insensibilité aux délais et temporisation

4.1 Introduction

L'instrumentation de la vérification des propriétés de sûreté et de vivacité est complètement indépendante du niveau de temporisation de notre modèle de circuits numériques asynchrones. En effet, les motifs de vérification de la sûreté et de la vivacité sont implantés au niveau des modules, alors que la temporisation du modèle est gérée au niveau du canal de communication point à point unidirectionnel. De plus, ces motifs de vérification s'inscrivent dans l'instant et non pas dans la durée. En effet, une erreur de sûreté est déclenchée à un instant donné au niveau d'un module. Dans le cas d'un blocage, la minuterie déclenche une occurrence d'événement à l'instant « blocage ». Cette occurrence est détectée au pas de simulation suivant, à l'instant « infini ».

Nous examinons soigneusement dans les paragraphes suivants la corrélation entre persistance et temporisation.

4.2 Vérification de la propriété de persistance et temporisation

La vérification de la propriété de persistance et le niveau de temporisation du modèle de circuits numériques asynchrones sont intrinsèquement liés. En effet, la propriété de persistance traduit l'atomicité d'une action. La notion d'atomicité d'une action est définie par rapport au modèle de temps. Comme on l'a vu, SystemC v2.0.1 propose 3 niveaux de temporisation. Ces niveaux sont réglables dans notre modèle de circuits numériques asynchrones au niveau des canaux de communication. On écarte d'emblée le niveau 0 de temporisation puisque dans ce cas, en présence du tirage aléatoire des processus, la simulation peut conduire à une rupture de la causalité à l'initialisation.

Etudions un exemple générateur-adaptateur-opérateur-consommateur en nous focalisant sur les notifications et les synchronisations. On peut voir cet exemple comme la chaîne de mise à zéro qu'on a introduite au chapitre IV. On regarde dans un premier temps le comportement du circuit non temporisé, puis le comportement du circuit temporisé. Dans le cas du circuit temporisé, on fait varier la temporisation du circuit de manière à faire apparaître différentes sources de violation de la persistance.

Nous sommes dans un schéma de production-consommation des occurrences d'événements. Suivant le protocole de communication unidirectionnel que nous avons défini au chapitre IV, chaque production verrouille le canal de communication et chaque consommation le déverrouille. Pour respecter la propriété de persistance, la production d'une requête à l'entrée du canal doit être plus lente que sa consommation à la sortie du canal. Mais, que se passe-t-il quand production et consommation fonctionnent à la même vitesse ? Cela conduit-il systématiquement à une violation de la persistance ?

Quoi qu'il en soit, nous nous attendons à ce qu'une erreur de persistance soit déclenchée lorsqu'un processus produit deux requêtes consécutives sur le même canal.

Pour raisonner, on présente sur la Figure 44 un diagramme simplifié de l'algorithme de simulation. Sur ce diagramme, on fait apparaître la phase d'initialisation où tous les processus sont activés dès la première évaluation. On fait abstraction du type de co-routine, on supprime la phase de mise à jour des canaux élémentaires et on ne tient pas compte du niveau 0 reservé à la sensibilisation dynamique des processus.

Dans chaque cas, on déroule l'algorithme de simulation à la main. Cela nous permet d'établir l'échéancier de la simulation, illustrant ainsi précisément la relation entre persistance et temporisation. Pour chaque processus, on note « synch » une synchronisation avec le noyau de simulation, « prod(req) » la notification d'une requête vers une sortie, « synch* » la sensibilisation d'un processus suite à une occurrence d'événement (requête ou acquittment), « cons(req) » la consommation d'une requête depuis une entrée, « prod(acq) » la notification d'un acquittement vers une entrée, et « cons(acq) » la consommation d'un acquittement depuis une sortie. Des flèches indiquent la relation entre la production d'une requête et la sensibilisation d'un processus.

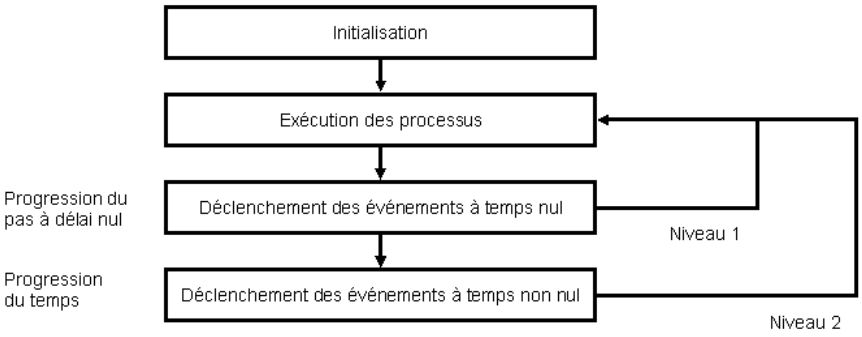


Figure 44 : Diagramme simplifié de l'algorithme de simulation de SystemC v2.0.1

4.3 Générateur – adaptateur – opérateur – consommateur

4.3.1 Spécification

Sur le diagramme de la Figure 45, on schématise le comportement d'un circuit générateur-adaptateur-opérateur-consommateur en mettant en évidence les notifications d'occurrences d'événements et les synchronisations entre les différents processus. Chaque processus est encapsulé dans une boucle sans fin ; ce dont on fait abstraction sur le diagramme. Les canaux de communication point à point unidirectionnels entre les modules sont temporisés par les variables de délais dg_req, dg_acq, da_req, do_req.

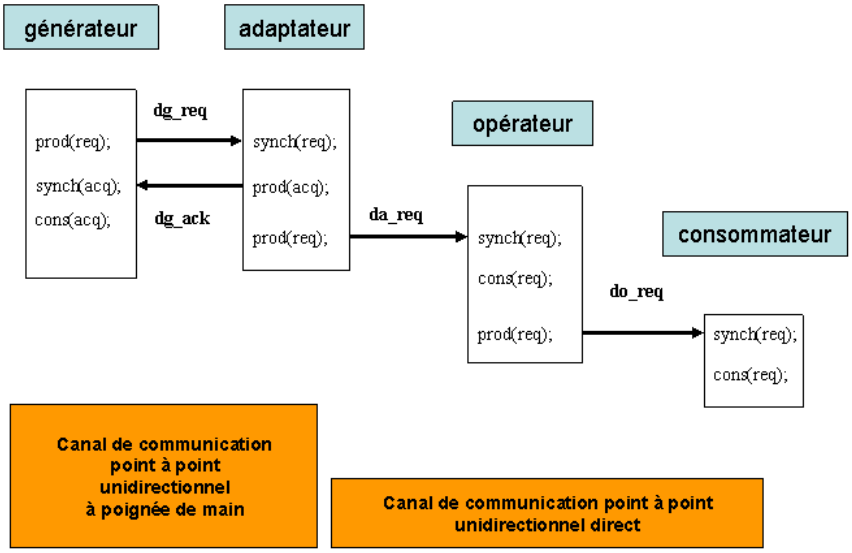


Figure 45 : Modèle du circuit générateur-adaptateur-opérateur-consommateur à temporisation variable

4.3.2 Comportement du modèle non temporisé

On pose :

- dg_req = délai_nul
- dg_acq = délai_nul

- da_req = délai_nul
- do_req = délai_nul

Cela correspond à un modèle non temporisé. L'exécution de l'algorithme de simulation nous conduit à l'échéanchier de la Figure 46.

Ordonnancement	Générateur	Adaptateur	Opérateur	Consomateur	Temps (pas à délai nul)
Evaluation n°1	prod(req) synch cons(acq)	synch	synch	synch	
Niveau 1	synch	synch*	synch	synch	1
Evaluation n°2	synch	cons(req) prod(acq) prod(req) synch	synch	synch	
Niveau 1	synch*	synch	synch*	synch	2
Evaluation n°3	prod(req) synch cons(acq)	synch	cons(req) prod(req) synch	synch	
Niveau 1	synch	synch*	synch	synch*	3

(a)

Ordonnancement	Générateur	Adaptateur	Opérateur	Consommateur	Temps (pas à délai nul)
Evaluation n°4	synch	cons(req) prod(acq) prod(req) synch	synch	cons(req) synch	
Niveau 1	synch*	synch	synch*	synch	4
Evaluation n°5	prod(req) synch cons(acq)	synch	cons(req) prod(req) synch	synch	
Niveau 1	synch	synch*	synch	synch*	5

(b)

Figure 46 : Echéancier de la simulation du circuit générateur-adaptateur-opérateur-consommateur non temporisé

La première évaluation concerne la phase d'initialisation. La 5ième évaluation est identique à la 3ième phase d'évaluation. On obtient alors un point fixe. La simulation débute par une production d'occurrence d'événement. Tout au long de la simulation, on n'obtient jamais de situation où soit un processus produit deux requêtes consécutives sur le même canal.

La propriété de persistance est respectée pour ce circuit non temporisé.

4.3.3 Comportement du modèle temporisé

Lorsque l'on introduit des délais dans le modèle, on change de modèle de temps. On passe d'un modèle de temps ordonné à un modèle de temps métré. Ce faisant, on change la nature de la notion d'atomicité. Qu'advient-il de la propriété de persistance ? Les configurations de délais respectent-elles toutes la propriété de persistance ?

Clairement non, puisqu'étant ouvert, le circuit ne peut être insensible aux délais. On distingue alors trois types de configuration de délais vis-à-vis de la propriété de persistance en présence du tirage aléatoire des processus : les configurations incorrectes, les configurations correctes et les configurations métastables.

4.3.3.1 Cas n°1 : Configurations incorrectes

On temporise le générateur de manière à produire des stimuli plus vite que le reste du circuit ne peut les consommer. On pose :

- dg_req = délai_nul
- dg_acq = 9 ps
- da_req = 10 ps
- do_req = 10 ps

L'exécution de l'algorithme de simulation nous conduit à l'échéanchier de la Figure 47.

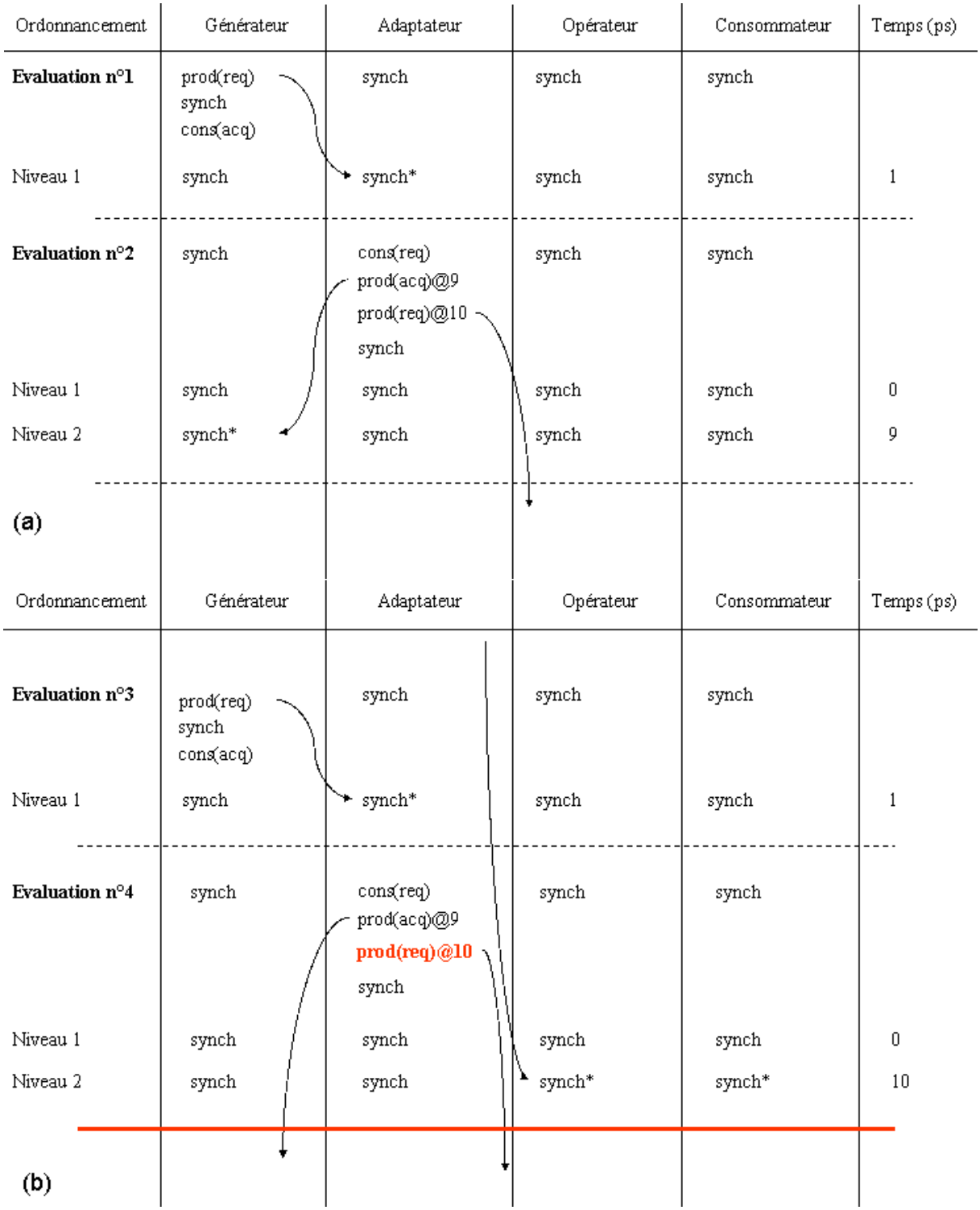


Figure 47 : Echéancier de la simulation du circuit générateur-adaptateur-opérateur-consommateur temporisé : configuration incorrecte

A la 2nde évaluation, l'adaptateur produit une requête sur le canal reliant l'adaptateur et l'opérateur. A la 4ième évaluation, l'adaptateur produit une requête sur le même canal. Entre temps, il n'y a pas eu de consommation de la requête de la part de l'opérateur.

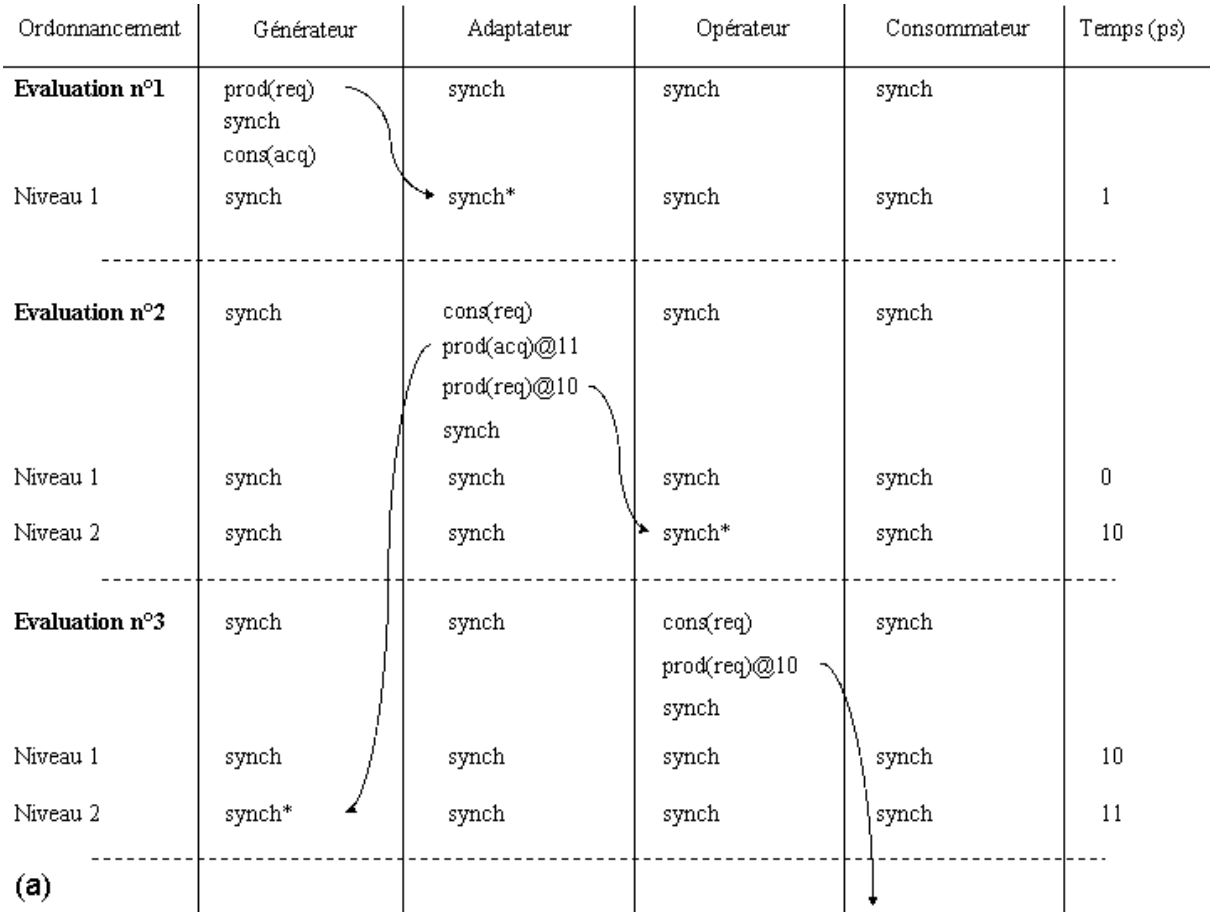
La propriété de persistance n'est jamais respectée pour cette configuration de délais du circuit.

4.3.3.2 Cas n°2 : Configurations correctes

On temporise le générateur de manière à produire des stimuli moins vite que le reste du circuit ne peut les consommer. On pose :

- dg_req = délai_nul
- dg_acq = 11 ps
- da_req = 10 ps
- do_req = 10 ps

L'exécution de l'algorithme de simulation nous conduit à l'échéanchier de la Figure 48.



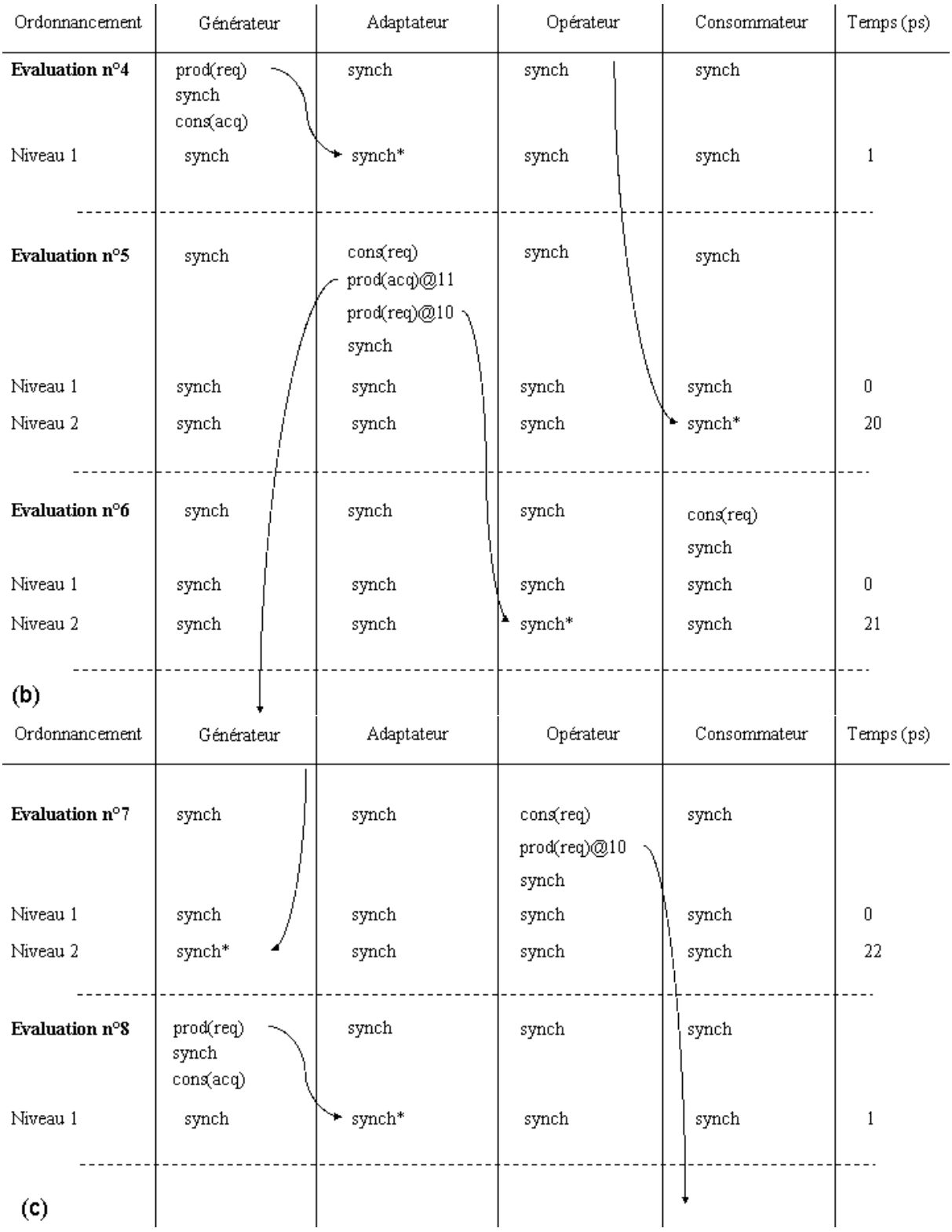


Figure 48 : Echéancier de la simulation du circuit générateur-adaptateur-opérateur-consommateur temporisé : configuration correcte

La 8^{ième} évaluation est identique à la 4^{ième} évaluation. On obtient alors un point fixe. La simulation débute par la production d'une requête. Tout au long de la simulation, on n'obtient jamais de situation où un processus produit deux requêtes consécutives sur le même canal.

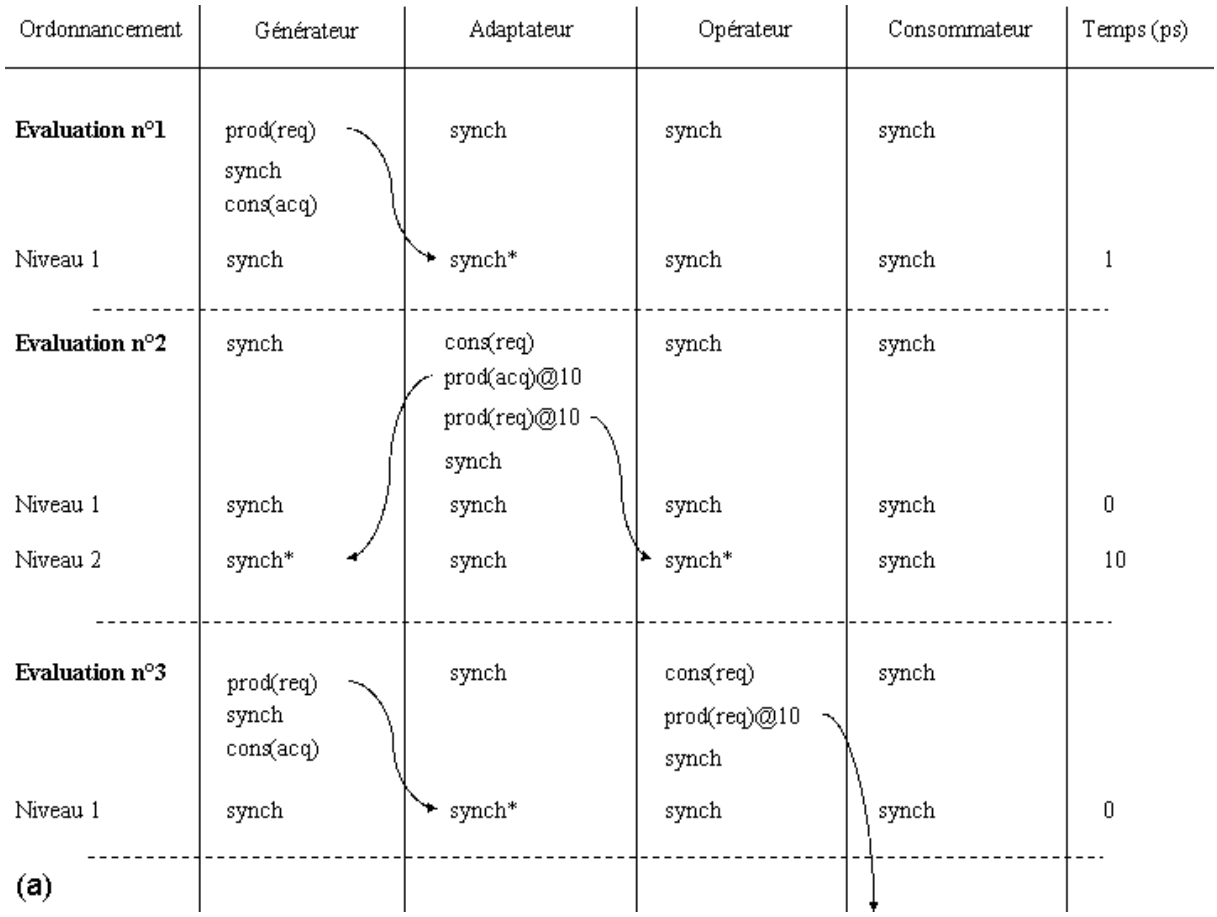
La propriété de persistance est respectée pour cette configuration de délais du circuit.

4.3.3.3 Cas n°3 : Configuration métastable

On temporise le générateur de manière à produire des stimuli à la même vitesse que le reste du circuit peut les consommer. On pose :

- dg_req = délai_nul
- dg_acq = 10 ps
- da_req = 10 ps
- do_req = 10 ps

L'exécution de l'algorithme de simulation nous conduit à l'échéanchier de la Figure 49.



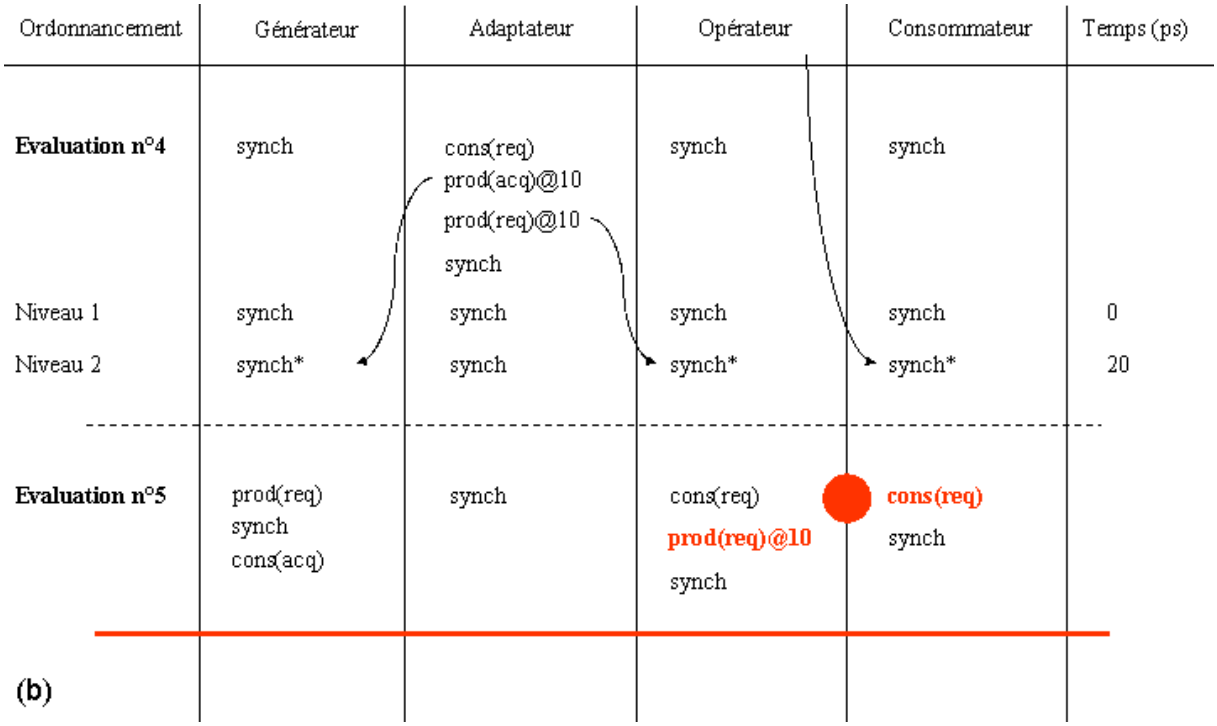


Figure 49 : Echéancier de la simulation du circuit générateur-adaptateur-opérateur-consommateur temporisé : configuration métastable

Au niveau de la 3ième évaluation, l'opérateur produit une requête, verrouillant ainsi le canal de communication point à point unidirectionnel entre l'opérateur et le consommateur.

Au niveau de la 5ième évaluation, la production d'une requête par l'opérateur et sa consommation par le consommateur sont activées en parallèle de part et d'autre du canal qui les relie.

Si le consommateur est exécuté en premier, il libère le canal de communication. L'opérateur peut alors verrouiller le canal.

Si l'opérateur est exécuté en premier, le canal ayant déjà été vérouillé à la 3ième évaluation, une erreur de persistance est détectée.

Le circuit se comporte de façon métastable. Le respect de la propriété de persistance dépend de l'ordre d'exécution des processus. L'ordre d'exécution des processus étant aléatoire, la violation de la propriété de persistance est détectée au bout d'un certain nombre de phases d'évaluation, ce nombre étant d'autant plus faible que le tirage aléatoire est de « bonne qualité ».

4.4 Résumé

La propriété de persistance repose sur la notion d'atomicité d'une action. Elle est donc fortement liée au modèle de temps utilisé pour simuler le système. Dans cette étude, nous avons mis en évidence que le passage du mode non temporisé au mode temporisé était délicat dans le cas d'un exemple de circuit sensible aux délais. On a identifié 4 types de comportements sur cet exemple en construisant l'échéancier de la simulation à partir d'un algorithme simplifié d'ordonnancement du noyau de simulation de SystemC v2.0.1.

En l'absence de temporisation, le comportement est systématiquement correct. En présence de temporisation, soit le comportement est systématiquement incorrect, soit le comportement est systématiquement correct, soit le comportement est métastable.

A chaque fois que le comportement est incorrect, on est capable de détecter la violation de la propriété de persistance grâce au mécanisme de verrou implanté dans le canal de communication point à point unidirectionnel.

Cela nous permet de conclure que l'instrumentation de la vérification de la propriété de persistance est indépendante du type de temporisation du modèle.

On arrive à la même conclusion pour les motifs de vérification des propriétés de sûreté et de vivacité, qui s'inscrivent dans l'instant et non dans la durée.

De plus, ces 3 motifs de vérification sont largement indépendants les uns des autres puisque la vérification de la persistance concerne les canaux de communication, la vérification de la sûreté concerne les modules et la vérification de la vivacité concerne les points de synchronisation des processus et les deux derniers pas de simulation.

Ainsi instrumenté, notre modèle respecte la sémantique des systèmes à événements discrets asynchrones. Il respecte le flot de conception descendant et notamment la phase de temporisation.

5 Mise en œuvre

5.1 Introduction

On a présenté l'oscillateur de Muller au paragraphe IV8. C'est a priori un circuit insensible aux délais à l'exception de la phase de mise à zéro du circuit. On a implanté dans ce circuit un générateur de stimuli permettant d'effectuer cette mise à zéro. Du coup le circuit (voir Figure 38) n'est plus insensible aux délais.

Nous allons utiliser cet exemple pour valider l'instrumentation de la vérification des propriétés de persistance et de vivacité de notre modèle de circuits numériques asynchrones. A moins d'un problème de conception, ce circuit est sûr car toutes les gardes des portes sont statiquement mutuellement exclusives.

Par conséquent, on crée artificiellement un exemple dont on sait pertinemment qu'il est non sûr. Cet exemple, le générateur de jetons est mis à contribution pour valider l'instrumentation de la vérification de la propriété de sûreté.

5.2 Oscillateur de Muller

5.2.1 Modélisation de niveau 0, simulation et vérification de la vivacité

On lance la simulation en activant la détection des blocages.

Dans le cas où le tirage aléatoire des processus n'est pas activé, on obtient systématiquement la trace suivante.

Pseudo code du modèle SystemC v2.0.1 non temporisé de niveau 0 de l'oscillateur de Muller et détection d'un blocage :

```
1  // début de la simulation
2  processus : raz.generator : écriture: 1
3  processus : raz.adaptor : initialisation
4  processus : fraz : initialisation
5  processus : p0 : initialisation
6  processus : p1: initialisation
7  processus : t0 : initialisation
8  processus : p2 : initialisation
9  processus : t1 : initialisation
10 Blocage détecté
11 // fin de la simulation
```

L'exécution n'est pas correcte et la simulation se termine en déclenchant une erreur de blocage comme on s'y attendait.

Dans le cas où le tirage aléatoire des processus est activé, soit la simulation donne un résultat correct et ne se termine pas, soit la simulation donne un résultat incorrect et on obtient la trace précédente.

L'instrumentation du modèle pour la vérification de la propriété de vivacité est ainsi validée.

5.2.2 Modélisation non temporisée, simulation et vérification de la persistance

On retrouve les résultats des paragraphes IV8.3 et IV8.4. Il n'y a pas de violation de la propriété de persistance à ce niveau de temporisation, comme on s'y attendait.

5.2.3 Modélisation temporisée, simulation et vérification de la persistance

5.2.3.1 Temporisation à délai unitaire

Puisque a priori le circuit est insensible aux délais, le circuit doit fonctionner correctement quelque soit la distribution des délais dans le modèle. On introduit donc des délais unitaires sur tous les canaux du circuit. Chaque canal de communication point à point unidirectionnel se voit affecté un délai de 10 ps pour une transition montante aussi bien que pour une transition descendante ; ceci est représenté sur la Figure 50.

Or, on a vu au paragraphe 4.3.3.3 que ce type de temporisation dans une chaîne producteur-opérateur-consommateur conduisait à un comportement métastable et à une violation de la propriété de persistance.

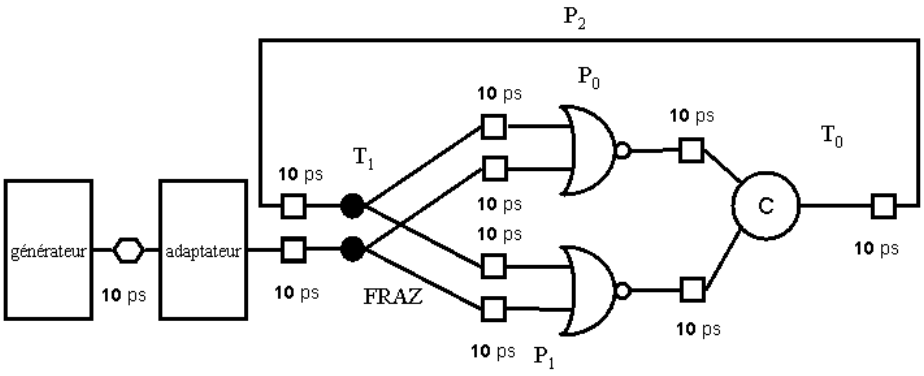


Figure 50 : Circuit de Patil de l'oscillateur de Muller avec mise à zéro et temporisation unitaire

5.2.3.2 Simulation et vérification de la persistance

Dans le cas où le tirage aléatoire des processus n'est pas activé, l'exécution n'est pas correcte et la simulation se termine au bout de 20 ps en déclenchant une erreur de persistance ; ce que montre la trace d'exécution qui suit. On retrouve le comportement explicité au paragraphe 4.3.3.1.

Pseudo trace d'exécution du modèle SystemC v2.0.1 temporisé de l'oscillateur de Muller et détection d'une erreur de persistance :

```
1  // début de la simulation
2  Processus p2:              @ 0 ps @    initialisation
3  Processus t0:              @ 0 ps @    initialisation
4  Processus raz.adaptor:     @ 0 ps @    initialisation
5  Processus raz.generator:   @ 0 ps @    écriture:   1
6  Processus t0:              @ 0 ps @    initialisation
7  Processus p0:              @ 0 ps @    initialisation
8  Processus p1:              @ 0 ps @    initialisation
9  Processus fraz:            @ 0 ps @    initialisation
10 Processus p0:              @ 10 ps @   initialisation
11 Processus t1:              @ 10 ps @   initialisation
12 Processus p1:              @ 10 ps @   initialisation
13 Processus raz.adaptor:     @ 10 ps @   lecture:    1
14 Processus raz.adaptor:     @ 10 ps @   écriture:   1
15 Processus fraz:            @ 10 ps @   lecture:    1
```

```
16 Processus fraz:            @ 20 ps @   écriture:   1
17 Processus fraz:            @ 20 ps @   écriture:   1
18 Processus raz.generator:   @ 20 ps @   écriture:   0
19 Processus raz.adaptor:     @ 20 ps @   lecture:    0
20 Processus raz.adaptor:     @ 20 ps @   écriture:   0
21 Processus p0:              @ 20 ps @   lecture:    1
22 Processus p0:              @ 20 ps @   lecture:    0
23 Processus p0:              @ 20 ps @   écriture:   0
24 Processus p0:              @ 20 ps @   lecture:    1
25 Processus p0:              @ 20 ps @   lecture:    0
26 Processus fraz:            @ 20 ps @   lecture:    0
27 Processus fraz:            @ 20 ps @   écriture:   0
28 Processus fraz:            @ 20 ps @   écriture:   0
29 Non persistance détectée
30 // fin de la simulation
```

Dans le cas où le tirage aléatoire des processus est activé, soit la simulation donne un résultat correct, soit on obtient la trace précédente.

Le temps nécessaire à initialiser le circuit vaut 40 ps. C'est le temps nécessaire pour traverser le générateur, l'adaptateur, la fourche d'initialisation et les portes « non-ou ».

Avec la configuration de délais choisie, on vérifie aisément que la demi-période d'oscillation du circuit vaut 40 ps.

Lorsque l'on lance la simulation pour une durée de 1001 ps, la simulation s'arrête exactement au bout de 1 ns. Cela correspond à la latence d'initialisation du circuit (40 ps) plus 12 périodes d'oscillation de période 80 ps comme le montre le chronogramme de la Figure 51.

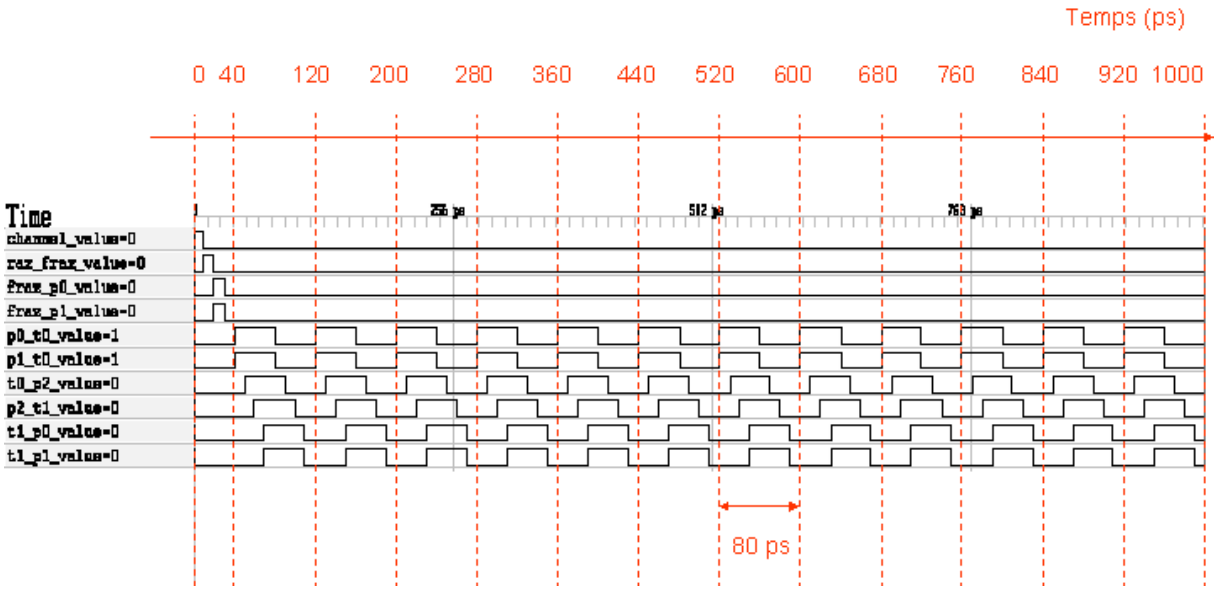


Figure 51 : Chronogramme de simulation de l'oscillateur de Muller temporisé à délais unitaires

5.2.3.3 Retemporisation, simulation et vérification de la propriété de persistance

Conformément au comportement explicité au paragraphe 4.3.3.2, on modifie la temporisation de la chaîne de mise à zéro pour retrouver un comportement systématiquement correct. Le délai d'acquittement du générateur de stimuli passe de 10 ps à 11 ps. On lance la simulation pour une durée de 1002 ps, la simulation s'arrête exactement au bout de 1001 ps. Cette durée correspond à la latence d'initialisation du circuit, soit 41 ps suivie de 12 périodes d'oscillation à 80 ps.

Le comportement obtenu est systématiquement correct et ne dépend plus du tirage aléatoire des processus. Il n'y a plus de violation de la propriété de persistance. Le chronogramme obtenu est similaire à celui de la Figure 51.

L'instrumentation du modèle pour la vérification de la propriété de persistance est ainsi validée.

5.3 Générateur de jetons

5.3.1 Spécification

Considérons la spécification sous forme de réseau de Petri ordinaire de la Figure 52. Il s'agit d'un générateur de jetons. A chaque oscillation, un nouveau jeton est introduit dans le réseau. Dès l'exécution de la première transition, la place P_1 contient 2 jetons et la propriété de sûreté est violée. C'est ce dont nous avons besoin pour valider l'instrumentation de la vérification de la propriété de sûreté.

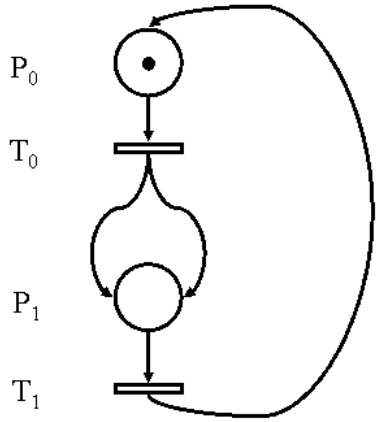


Figure 52 : Réseau de Petri ordinaire du générateur de jetons

5.3.2 Traduction de Patil sur événement

Pour faire apparaître une erreur de sûreté dans le contexte de la modélisation de circuits numériques asynchrones, il faut recourir à des portes dont les gardes ne sont pas statiquement mutuellement exclusives. Sinon, tout ce que l'on parvient à détecter se ramène à une erreur de persistance. Or dans le contexte de la conception des circuits numériques, on évite systématiquement ce cas.

Il nous faut donc sortir du cadre que nous nous sommes fixé pour valider notre instrumentation. La traduction de Patil reste valable mais on peut raisonner sur des portes effectuant des calculs non sur des valeurs comme nous l'avons supposé jusqu'ici, mais sur des

événements. Ce point se ramène aux problèmes de la puissance d'expression de SystemC v2.0.1 et de la séparation entre calcul et communication que nous avons étudiés au chapitre V.

Nous ne disposons donc plus ici de l'opérateur logique de négation, ni de la possibilité de programmer des formes normales disjonctives pour les gardes. Les variables booléennes ne traduisent plus des valeurs, mais des transitions. Les règles de production, ou plutôt devrait-on dire les règles de production-consommation deviennent :

- $p \rightarrow q$: une transition sur p donne une transition sur q,
- $p \rightarrow q_0, q_1$: une transition sur p donne une transition sur q_0 et une transition sur q_1,
- $p_0 \vee p_1 \rightarrow q$: une transition sur p_0 ou une transition sur p_1 donnent une transition sur q,
- $p_0 \wedge p_1 \rightarrow q$: une transition sur p_0 et une transition sur p_1 donnent une transition sur q.

Ces règles de traduction nous conduisent au circuit de la Figure 53. Ce circuit est composé d'un générateur, d'un adatateur de protocole, d'une porte « ou », d'une fourche, d'une porte « ou exclusif », de fils et de canaux. La porte « ou exclusif » représentant la place P_3 dans le circuit est instrumentée pour détecter la violation de la propriété de sûreté.

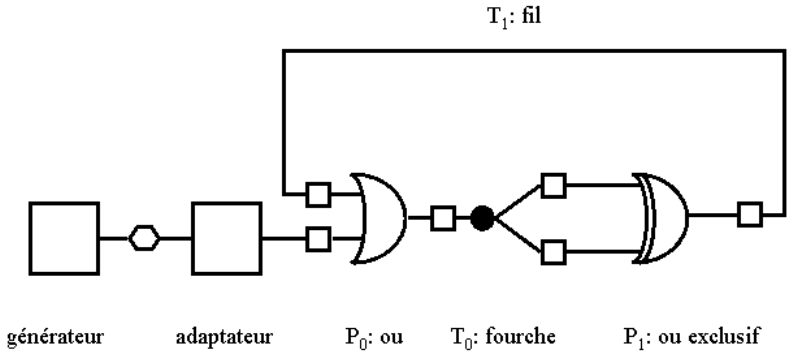


Figure 53 : Circuit de Patil sur événement du générateur de jeton avec initialisation

Le pseudo code de cet exemple se trouve reporté en annexe (voir D.4).

5.3.3 Modélisation non temporisée, simulation et vérification de la sûreté

L'exécution de l'algorithme de simulation nous conduit à l'échéanchier de la Figure 54.

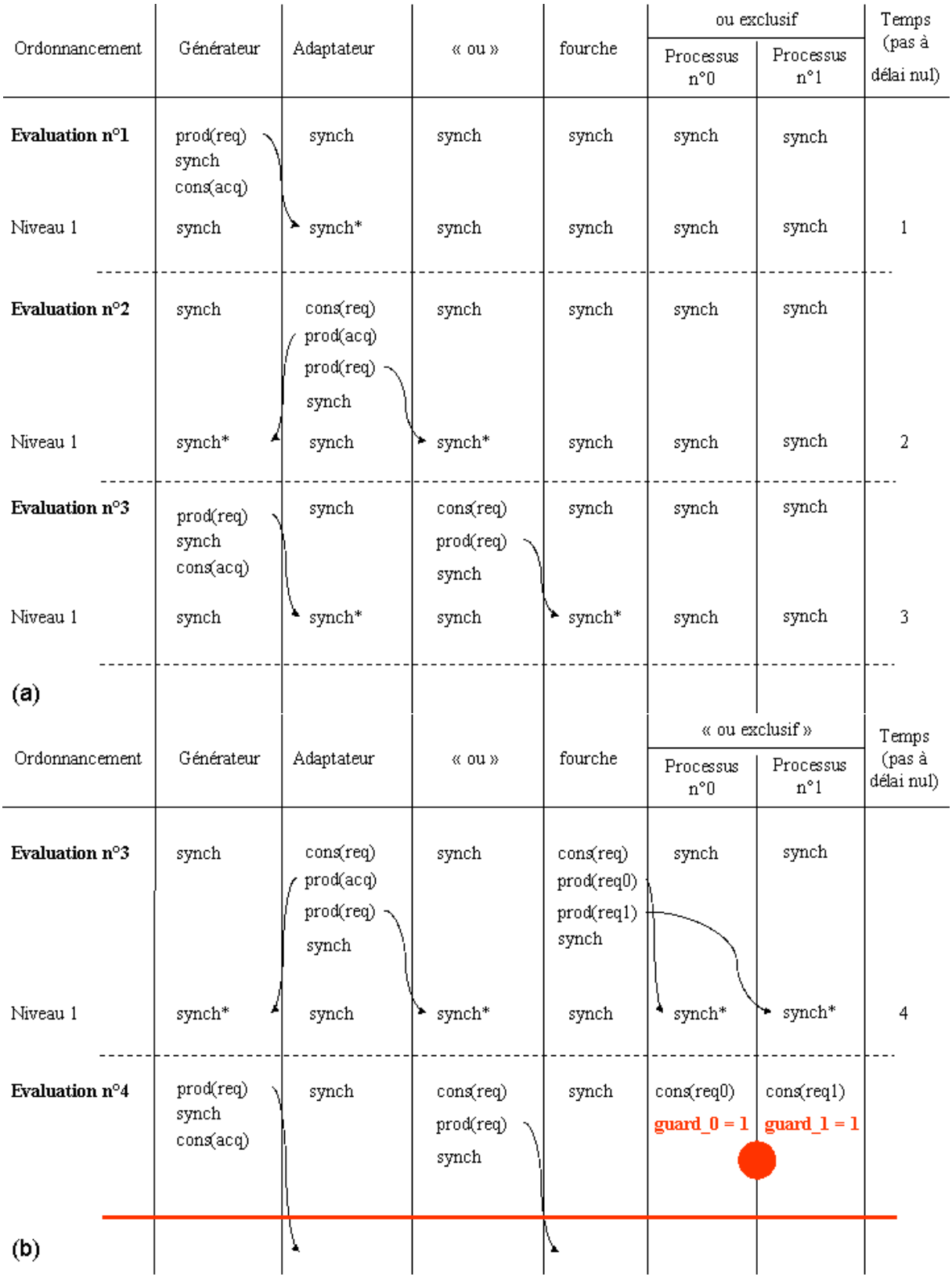


Figure 54 : Echéancier de la simulation du générateur de jetons non temporisé

Au bout de la 4ième évaluation, les deux processus de la porte « ou exclusif » sont évalués en parallèle. Leurs gardes passent toutes deux à 1. Cela est détecté dans le calcul du dernier processus de la porte en cours d'exécution et conduit à l'arrêt de la simulation.

La simulation s'interrompt effectivement au 4ième pas de calcul à délai nul comme le montre le chronogramme de la Figure 55. Suivant l'ordre d'exécution des processus en fonction du tirage aléatoire, soit le processus n°0, soit le processus n°1 déclenche l'arrêt de la simulation.

L'instrumentation du modèle pour la vérification de la propriété de sûreté est ainsi validée.

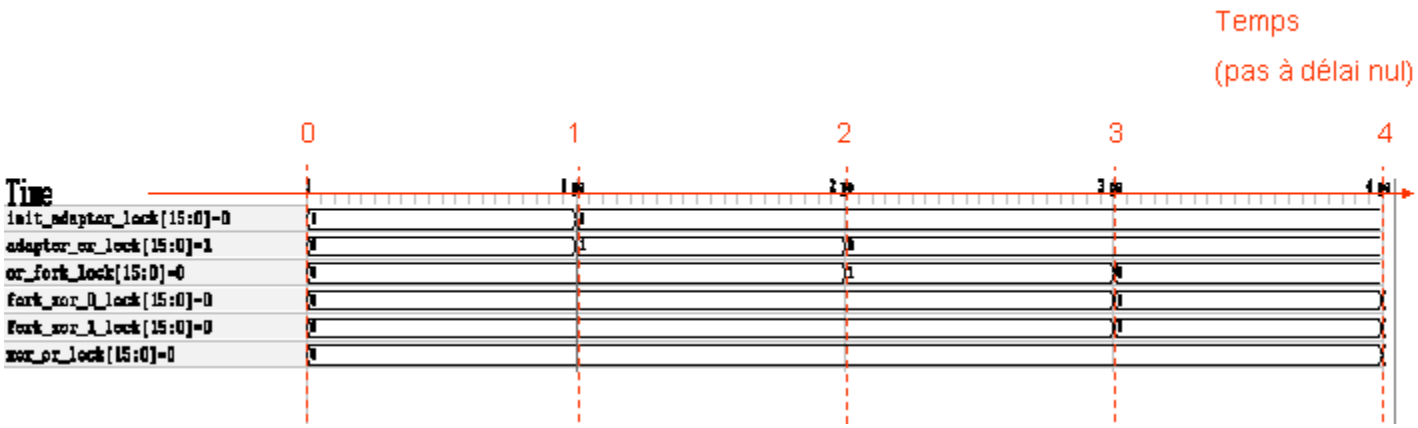


Figure 55 : Chronogramme de simulation du générateur de jetons non temporisé

6 Résultats

Finalement, nous obtenons un modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais, s'intégrant dans le standard SystemC v2.0.1 de conception de systèmes numériques. Cette instrumentation respecte la sémantique du modèle initial et notamment le procédé de temporisation. On complète le modèle en 4 couches du chapitre IV comme le montre la Figure 56.

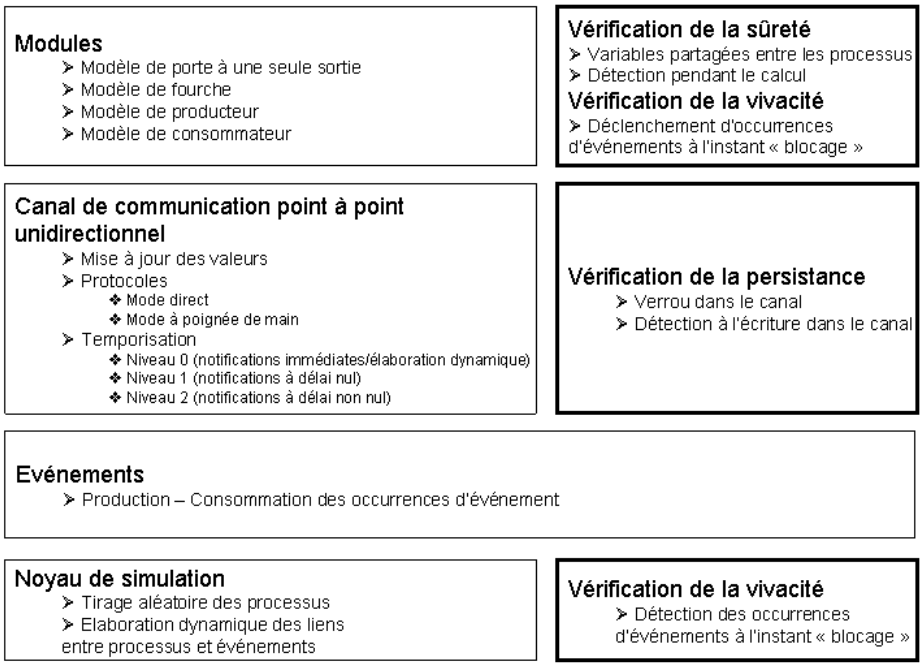


Figure 56 : Modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais, s'intégrant dans le standard SystemC v2.0.1 de conception de systèmes numériques

La vérification de la persistance intervient au niveau du canal de communication point à point unidirectionnel. Il s'agit de lui conférer le comportement d'un verrou. Dès lors, il est possible de détecter si une communication est en cours avant d'enclencher une nouvelle communication. Dans le cas où le canal n'est pas libre pour une nouvelle communication, on détecte une violation de la persistance et on déclenche l'arrêt de la simulation.

La vérification de la sûreté intervient au niveau des portes. Il s'agit de conférer à une porte le comportement d'un verrou. Alors, il est possible de détecter si deux processus sont activés simultanément. Dans le cas où les deux processus d'une porte se retrouvent actifs simultanément, on détecte une violation de la sûreté et on arrête la simulation.

Seule la vérification de la vivacité concerne à la fois le noyau de simulation et les portes du modèle. Il s'agit d'identifier un blocage. On modifie alors la sémantique des deux derniers pas de simulation et on introduit la détection d'un blocage dans le noyau de simulation. On déclenche un blocage à l'instant « blocage » lorsqu'une occurrence d'événement produite à un instant donné n'est pas consommée au bout d'un temps infini. Dés qu'un blocage est détecté, on déclenche l'arrêt de la simulation.

Nous mettons ces résultats en œuvre au paragraphe 5 sur deux exemples de circuits numériques asynchrones : l'oscillateur de Muller et le générateur de jetons. Cela nous permet de valider l'instrumentation du modèle pour la vérification des propriétés d'insensibilité aux délais, soit persistance, sûreté et vivacité.

Ainsi nous adressons les quatre premières phases du flot descendant de conception des circuits numériques asynchrones dans le standard SystemC v2.0.1 de conception de systèmes numériques :

- modélisation non temporisée,
- simulation non temporisée et vérification des propriétés d'insensibilité aux délais,
- temporisation du modèle,
- simulation temporisée et vérification des propriétés d'insensibilité aux délais.

Conclusion

Dans un premier temps, nous présentons les résultats de l'étude que nous venons de mener à bien. Nous mettons en évidence la contribution qui en découle, tant sur le plan théorique, que sur le plan pratique. Dans un second temps, nous élargissons la perspective, qui se révèle multiple.

Etude et contribution

Nous venons d'étudier sur le plan théorique les circuits numériques asynchrones en tant que sous classe des systèmes à événements discrets asynchrones. Nous venons d'établir que pour spécifier un circuit numérique asynchrone par un programme faisant abstraction des délais, il fallait et il suffisait que le circuit vérifiât trois propriétés fondamentales, propriétés que nous appelons propriétés d'insensibilité aux délais, i.e. persistance, sûreté et vivacité. Ce théorème constitue la partie théorique de la contribution de cette thèse.

Nous venons de choisir le standard SystemC v2.0.1 de conception de systèmes numériques pour élaborer notre modèle de circuits numériques asynchrones. Ce standard supporte déjà la sémantique des systèmes à événements discrets synchrones. Nous venons d'adapter la sémantique de ce langage à la sémantique des systèmes à événements discrets asynchrones. Puis, nous avons bâti un modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais, i.e. persistance, sûreté et vivacité, s'intégrant dans un standard de conception de systèmes numériques. Ce modèle, représenté sur la Figure 57 correspond à la partie pratique de la contribution de cette thèse.

Décrivons successivement les différentes couches de ce modèle.

Le premier niveau concerne la couche « noyau de simulation » de SystemC v2.0.1. La modification permet de supporter la sémantique des systèmes à événements discrets asynchrones en intégrant le tirage aléatoire des processus dans le noyau. Dans ce contexte, on a interdit le recours à l'ordonnancement de niveau 0, réservé à la sensibilisation dynamique des processus.

C'est à ce premier niveau qu'intervient la vérification de la vivacité. On modifie la sémantique des deux derniers pas de simulation qu'on transforme en instant « blocage » et instant « infini ». On introduit la détection d'un blocage dans le noyau de simulation. Dès qu'un blocage est détecté, on arrête la simulation.

Le second niveau concerne la couche « événements » de SystemC v2.0.1. La modification consiste à introduire un couplage minimal entre noyau de simulation et modèle pour augmenter la puissance d'expression de l'algèbre de synchronisation du langage. Un modèle SystemC v2.0.1 et le noyau de simulation communiquent alors suivant un protocole de production-consommation des occurrences d'événements. La conséquence est la possibilité d'implémenter des structures de choix et ainsi d'élever la puissance d'expression de l'algèbre sur événement de SystemC v2.0.1 au niveau des réseaux de Petri ordinaires.

Le troisième niveau concerne la couche « communication » de SystemC v2.0.1. La modification consiste à introduire un nouveau canal de communication point à point unidirectionnel. Ce canal supporte la mise à jour des valeurs suivant le protocole de production-consommation des occurrences d'événements introduit au niveau « événements ».

Il supporte deux protocoles de communication : le mode direct et le mode à poignée de main. Il supporte également 3 niveaux de temporisation : le niveau 0 ou immédiat, le niveau 1 ou à délai nul et le niveau 2 ou à délai non nul. Bien que réservé à la sensibilisation dynamique des processus, le niveau 0 reste activable au niveau du canal.

C'est à ce troisième niveau qu'intervient la vérification de la persistance. Il s'agit de conférer au canal le comportement d'un verrou. Dès lors, il est possible de détecter si une communication est en cours avant d'enclencher une nouvelle communication. Dans le cas où le canal n'est pas libre pour une nouvelle communication, on détecte une violation de la persistance. Cela déclenche l'arrêt de la simulation.

Le quatrième niveau concerne la couche « modèles de calcul » de SystemC v2.0.1. La modification consiste à introduire les modèles spécifiques aux circuits numériques asynchrones à partir des circuits numériques asynchrones formalisés en LTL. On trouve un modèle de porte à une seule sortie et un modèle de fourche. Pour résoudre le problème de l'initialisation de la simulation en l'absence d'une horloge globale et par souci de complétude, on introduit un producteur et un consommateur dans le modèle de calcul. La porte à une seule sortie, la fourche et le consommateur s'appuient sur le protocole de communication point à point direct. Le producteur est un modèle hiérarchique. Il se divise en deux modèles : un générateur et un adaptateur. Ces deux modèles sont reliés par un canal de communication point à point unidirectionnel à poignée de main. L'adaptateur réalise la conversion entre le protocole à poignée de main et le protocole direct.

C'est à ce quatrième niveau qu'intervient la vérification de la sûreté. Il s'agit de conférer à une porte le comportement d'un verrou partagé entre les processus de la porte. Alors, il est possible de détecter si deux processus sont activés simultanément. Dans le cas où les deux processus d'une porte se trouvent actifs simultanément, on détecte une violation de la sûreté. Cela arrête la simulation.

A ce niveau également, intervient la vérification de la vivacité. On déclenche une occurrence d'événement à l'instant « blocage » lorsqu'une occurrence d'événement produite à un instant donné n'est pas consommée à l'instant « infini ». On simule donc le modèle de calcul jusqu'à l'instant « infini » pour détecter les blocages éventuels et arrêter la simulation le cas échéant.

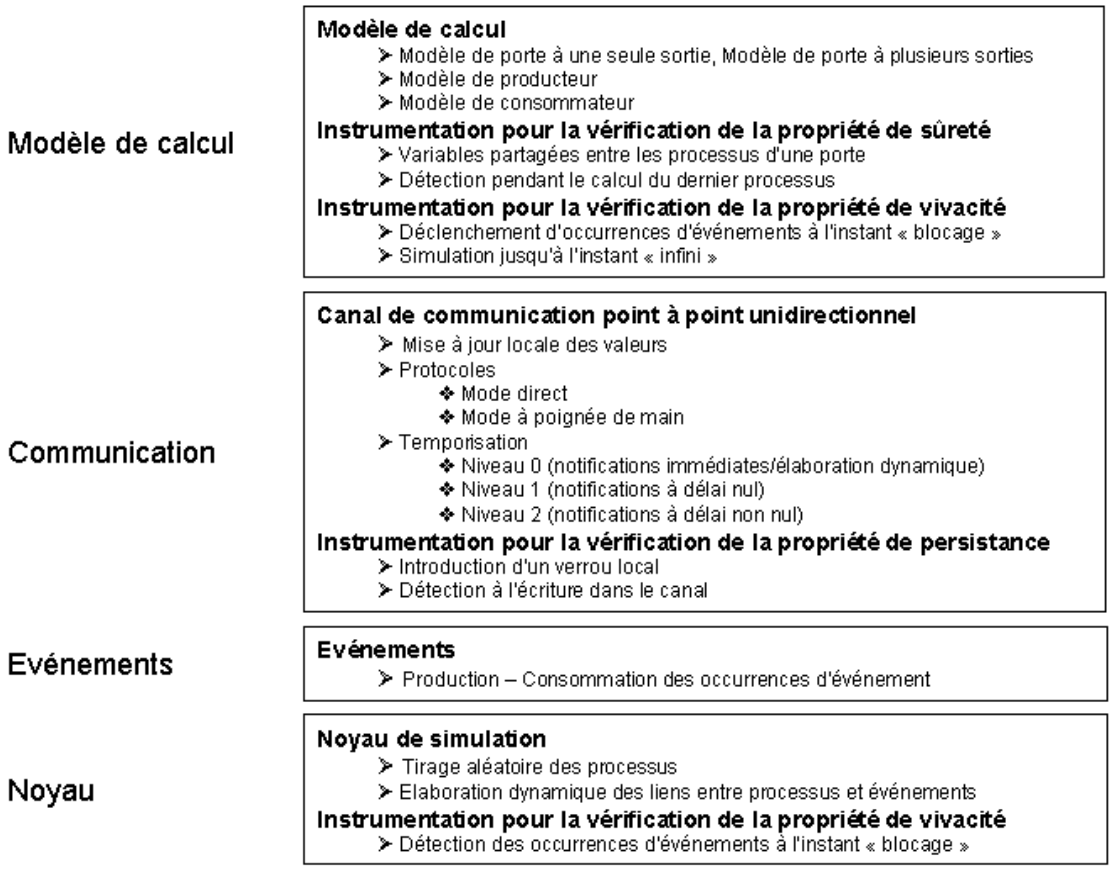


Figure 57 : Modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais, s'intégrant dans le standard SystemC v2.0.1 de conception de systèmes numériques

Ainsi nous adressons les quatre premières phases du flot descendant de conception des circuits numériques asynchrones dans le standard SystemC v2.0.1 de conception de systèmes numériques :

- modélisation non temporisée,
- simulation non temporisée et vérification des propriétés d'insensibilité aux délais,
- temporisation du modèle,
- simulation temporisée et vérification des propriétés d'insensibilité aux délais.

Notre objectif : établir un modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais, s'intégrant dans un standard de conception de systèmes numériques est atteint.

Perspective

Quatre directions se dégagent clairement dans la perspective. La première concerne la partie théorique et les trois suivantes la partie pratique de la contribution de cette thèse.

Sur le plan théorique, il faut étudier plus finement les tenants et les aboutissants du théorème sur l'abstraction des délais dans les systèmes à événements discrets asynchrones. Le recul permettra d'en mesurer la portée.

Sur le plan pratique, la première direction concerne le standard SystemC v2.0.1. Nous sommes aujourd'hui en mesure de proposer une évolution du standard SystemC v2.0.1 pour supporter les systèmes à événements discrets au sens large, i.e. aussi bien synchrones qu'asynchrones. La standardisation étant une démarche politique lourde de sens, il faut se tourner vers les décideurs pour organiser le support progressif de l'approche asynchrone par le standard SystemC v2.0.1 conformément aux recommandations de l'ITRS 2003.

La seconde direction concerne le modèle de circuits numériques asynchrones lui-même. Il y a beaucoup à faire pour en améliorer les caractéristiques. Par exemple, il faut travailler sur la généricité (types de données, ports, processus), l'interface avec d'autre modèles (niveau « sc_signal » dans SystemC v2.0.1 par exemple), les performances (améliorer les performances de la simulation, accélérer la convergence de la détection des erreurs), l'extension du champ de vérification à d'autres propriétés des systèmes à événements discrets (équité, contrôlabilité, observabilité) et la prise en compte d'aspects « analogiques » comme la consommation d'énergie ou la mise sous tension.

La troisième direction concerne la complétion du flot de conception. Cette étude s'est focalisée sur la modélisation, la simulation et la vérification des circuits numériques asynchrones dans le standard SystemC v2.0.1 de conception de systèmes numériques. Il faut compléter le flot en étudiant la vérification formelle, la synthèse et le test suivant la même approche.

Il reste donc beaucoup à faire pour élaborer un standard de conception de systèmes numériques en général, intégrant la conception des circuits numériques asynchrones en particulier.

Bibliographie

[1] D. Hilbert and P. Bernays, Grundlagen der Mathematik, 2 vols. Berlin, 1934-1939.

[2] Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, Monatshefte für Mathematik und Physik, vol. 38, 1931 : 173-198.

[3] Turing, A., On computable numbers with an application to the Entscheidungsproblem, Proc. Amer. Math. Soc., 43 (1937), 544-546.

[4] J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.

[5] W. Wolf, How many system architectures ?, IEEE Computer, pp 93-95, March 2003.

[6] The International Technology Roadmap for Semiconductors, 2003 edition, International Sematech, Austin, Texas, December 2003, http://public.itrs.net.

[7] M. Renaudin, Asynchronous circuits and systems: a promising design alternative, Elsevier Science Publishers B. V. Amsterdam, The Netherlands, 133 - 149, 2000.

[8] ISO/IEC 7498-1:1994, Information technology, Open Systems Interconnection, Basic Reference Model : The Basic Model, Geneva, Switzerland, 1994, http://www.iso.org.

[9] IEC 61691-1, Design automation, Part 1: VHDL language reference manual, Jul 1997, Geneva, Switzerland, http://www.iec.ch.

[10] IEEE 1364-1995 Verilog Language Reference Manual, 1995, New York, NY USA, http://www.ieee.org.

[11] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In Proceedings of an International Symposium on the Theory of Switching, pages 204-243. Harvard University Press, April 1959.

[12] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, Introduction to VLSI Systems, chapter 7. Addison-Wesley, 1980.

[13] J. Sparsø and S. Furber (eds.), Principles of asynchronous circuit design - A systems perspective. Kluwer Academic Publishers, 2001.

[14] C. J. Myers, Asynchronous Circuit Design, John Wiley and Sons, July 2001.

[15] Scott Hauck. Asynchronous design methodologies: An overview. Proceedings of the IEEE, 83(1):69-93, January 1995.

[16] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In XI Conference on Design of Integrated Circuits and Systems, Barcelona, November 1996.

[17] A. J. Martin, “The limitations to delay-insensitivity in asynchronous circuits,” in Advanced Research in VLSI (W. J. Dally, ed.), pp. 263–278, MIT Press, 1990.

[18] Rajit Manohar and Alain J. Martin, Quasi-Delay Insensitive Circuits are Turing-Complete. Invited paper, Async96 Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, March 1996.

[19] Ivan E. Sutherland. Micropipelines. Communications of the ACM, 32(6):720-738, June 1989.

[20] Ivan Blunno and Luciano Lavagno. Automated synthesis of micro-pipelines from behavioral Verilog HDL. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 84-92. IEEE Computer Society Press, April 2000.

[21] D. A. Huffman. The synthesis of sequential switching circuits. Journal of the Franklin Institute. 1954.

[22] S. H. Unger. Asynchronous Sequential Switching Circuits. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[23] W. S. Coates, A. L. Davis, and K. S. Stevens. "Automatic Synthesis of Fast Compact Asynchronous Control Circuits", IFIP Working Conference on Asynchronous Design Methodologies, March 1993.

[24] Yun, K. Y., Dill, D. L., and Nowick, S. M. Practical generalizations of asynchronous state machines. In Proc. European Conference on Design Automation (EDAC) (Feb. 1993), IEEE Computer Society Press, pp. 525--530.

[25] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[26] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K.G. Larsen, S. Skyum, and G. Winskel, editors, Proc. International Colloquium on Automata, Languages and Programming (ICALP'98), Lecture Notes in Computer Science 1443, pages 1--16. Springer, 1998.

[27] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers principles, techniques, and tools. AddisonWesley, Reading, MA, 1986.

[28] C. A. Petri, Fundamentals of a theory of asynchronous information flow, Proc. IFIP Conf. Munich, pp 166-168, 1962.

[29] T. Murata. Petri-Nets: Properties, Analysis and Applications. Proceedings of the IEEE, Vol. 77, No. 4, pages 541-580. April 1989.

[30] J. Esparza, M. Nielsen. Decidability Issues for Petri Nets|a Survey. Journal of Information Processing and Cybernetics, Vol. 30, No. 3, pp. 143-160, 1995.

[31] J. Esparza. Decidability and complexity of Petri net problems --- an introduction. In Advances in Petri Nets 1998, volume 1491 of Lecture Notes in Computer Science, pages 374--428. Springer, 1998.

[32] J. Esparza, J. Desel. Free Choice Petri Nets. Cambridge University Press, 1995.

[33] Nadia Busi, Petri Nets with Inhibitor and Read Arcs: Semantics, Analysis and Application to Process Calculi Dipartimento di Matematica, Universita' di Siena, February 1998

[34] A. Yakovlev, L. Gomes, L. Lavagno,. Hardware Design and Petri Nets. Kluwer Academic Publishers, 2000.

[35] M. Diaz, Les réseaux de Petri - Modèles fondamentaux, Informatique et Systèmes d'Information, Hermès science, Paris, 2001.

[36] David Misunas. Petri nets and speed independent design. Communications of the ACM, 16(8):474-481, August 1973.

[37] Patil S., Circuit Implementation of Petri Nets, Computation Structures Group Memo 73, Project MAC, MIT, Cambridge, Massachussets, December 1972.

[38] M. Hack, Analysis of production schemata by Petri nets, Project MAC, MIT, Cambridge, Massachussets, 1972.

[39] C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

[40] A. Mazurkiewicz. Trace theory. In Petri Nets: Applications and Relationships to Other Model of Concurrency, Advances in Petri nets 1986, Part II; Proceedings of an advanced Course, pages 279-324. Springer Verlag, LNCS 255, 1986.

[41] R. Milner. Communication and Concurrency. Prentice Hall, 1989.

[42] E. Best, COSY: Its Relation to Nets and CSP. In Petri Nets: Applications and Relationships to Other Model of Concurrency, Advances in Petri nets 1986, Part II; Proceedings of an advanced Course, pages 416-440. Springer Verlag, LNCS 255, 1986.

[43] U. Goltz, CCS and Petri nets, Semantics of Systems of Concurrent Processes, Lecture Notes in Computer Science, vol. 469, ed. I. Guessarian, pp.334—357, Springer, Berlin 1990.

[44] T. Agerwala, A complete model for representing the coordination of asynchronous processes, Hopkins Computer Research Report No. 32, Computer Science Program, John Hopkins Univ., August 1974.

[45] ISO/IEC 15437:2001, Information technology, Enhancements to LOTOS (E-LOTOS), Geneva, Switzerland, Sept 2001, http://www.iso.org.

[46] ITU-T, Recommandation Z.100, Specification and Description Language, Geneva, Switzerland, August 2002, http://www.itu.int.

[47] Tobias Bjerregaard, Jens Sparsø and Shankar Mahadevan, Jan Madsen, Modeling Asynchronous Communication at Different Levels of Abstraction Using SystemC v2.0.1, 3rd ACiD-WG Workshop, Jan 2003.

[48] Open SystemC Initiative (OSCI). http://www.systemc.org.

[49] SystemC v2.0.1 2.0.1 Language Rule Manual, revision 1.0, OSCI, http://www.SystemC v2.0.1.org, 2003.

[50] FUNCTIONAL SPECIFICATION for SystemC v2.0.1 2.0, Update for SystemC v2.0.1 2.0.1, Version 2.0-Q, OSCI, http://www.SystemC v2.0.1.org, April 2002.

[51] SystemC v2.0.1 Version 2.0 User's Guide, Update for SystemC v2.0.1 2.0.1, OSCI, http://www.systemc.org, 2002.

[52] SystemC v2.0.1, Master-Slave Communication Library, OSCI, http://www.systemc.org, 2002.

[53] SystemC v2.0.1 Standard Verification Specification, version 1.0c, SystemC v2.0.1 Verification Working Group, http://www.systemc.org, January, 2003.

[54] TestBuilder releases 0.92 to 1.3, document and source code available at http://www.testbuilder.net.

[55] Modeling Software with SystemC v2.0.1 3.0, Thörsten Grötker, 6th ESCUG Meeting, October 2002.

[56] SpecC System. http://www.ics.uci.edu/~specc.

[57] K. Einwich *et al.*, “Analog Mixed Signal Extensions for SystemC v2.0.1”, White paper and proposal for the foundation of an OSCI Working Group (SystemC-AMS working group), http://mixsigc.eas.iis.fhg.de/, June 2002.

[58] K. Einwich, SystemC-AMS Working Group, Mission, Objectives, Roadmap, http://mixsigc.eas.iis.fhg.de/, Sept. 2002.

[59] ISO C++ standards official site. http://std.dkuug.dk/jtc1/sc22/wg21.

[60] G. Martin, SystemC and the Future of Design Languages: Opportunities for Users and Research, SBCCI, p 61, Sept. 2003.

[61] M. E. Conway, Design of a Separable Transition-Diagram Compiler, Comm. ACM 6 (7), pp. 8–15, 1963.

[62] F. Herrera, P. Sanchez, E. Villar, Modeling of CSP, KPN, and SR Systems with SystemC, Forum on specification & Design Languages, 2003.

[63] W. Mueller, J.Ruf, D. Hofmann, J. Gerlach, T. Kropf, and W.Rosenstiehl, The Simulation Semantics of SystemC, In Design, Automation and Test in Europe, pages 64-70, 2001.

[64] R. Drechsler and D. Groe., Reachability analysis for formal verification of SystemC, In EUROMICRO, pages 337--340, 2002.

[65] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multivalued ar-automata. In Design, Automation and Test in Europe, pages 742--748, 2001.

[66] Jo C. Ebergen. Translating Programs into Delay-Insensitive Circuits. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1987.

[67] Erik Brunvand. Translating Concurrent Communicating Programs into Asynchronous Circuits. PhD thesis, Carnegie Mellon University, 1991.

[68] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In Proc. European Conference on Design Automation (EDAC), pages 384-389, 1991.

[69] A. Bardsley and D. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, Hardware Description Languages and their Applications (CHDL), pages 89-91, April 1997.

[70] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," J. Appl. Phys., vol. 19, no. 1, pp. 55--63, Jan. 1948.

[71] Formal Method Education Resources, http://www.cs.indiana.edu/formal-methods-education

Annexe

A Annexe du chapitre I

A.1 Description d'un système à événements discrets

Considèrons un système rudimentaire composé d'une machine et de son environnement décrit algébriquement dans le cadre de CSP.

Le système est composé de deux processus P et Q. Le processus P choisit entre deux actions T_0 ou T_1, puis effectue l'action T_2 et reboucle sur lui-même. Le processus Q effectue les actions T_0 puis T_1 et reboucle sur lui-même.

On considère que les actions T_0 et T_1 nécessitent la collaboration des processus P et Q pour se réaliser alors que l'action T_2 est une action interne à P et ne nécessite pas l'intervention du processus Q.

A.2 Spécification et composition déterministe dans CSP

Dans un premier temps, on suppose que le système est déterministe au sens de CSP et on cherche à calculer son comportement à l'aide de l'opérateur de composition parallèle déterministe que fournit CSP. On note $\rightarrow$ l'opérateur de composition séquentielle, | l'opérateur de disjonction déterministe, || l'opérateur de composition parallèle déterministe et $STOP$ le processus d'arrêt.

Le système s'écrit :

S_1 $\quad P = (T_0 \mid T_1) \rightarrow T_2 \rightarrow P$

S_2 $\quad Q = T_0 \rightarrow T_1 \rightarrow Q$

S_3 $\quad P \,\square\, Q = ((T_0 \mid T_1) \rightarrow T_2 \rightarrow P) \,\square\, (T_0 \rightarrow T_1 \rightarrow Q)$

CSP fournit les règles suivantes, qu'on appliquera dans ce contexte.

R_1 $\quad (P \mid Q) \rightarrow R = (P \rightarrow R) \mid (Q \rightarrow R)$

R_2 $\quad P \mid STOP = P$

R_3 $\quad (P \mid Q) \,\square\, R = (P \,\square\, R) \mid (Q \,\square\, R)$

On suppose que les événements c et d nécessitent la collaboration des processus P et Q et dans ce cas, la composition parallèle se réduit à :

R_4 $\quad (c \rightarrow P) \,\square\, (c \rightarrow Q) = c \rightarrow (P \,\square\, Q)$

R_5 $\quad (c \rightarrow P) \,\square\, (d \rightarrow Q) = STOP$

On suppose que les événements a et b ne nécessitent pas la collaboration des processus P et Q et dans ce cas, la composition parallèle se réduit à :

R_6 $\quad (a \rightarrow P) \,\square\, (c \rightarrow Q) = a \rightarrow (P \,\square\, (c \rightarrow Q))$

R_7 $\quad (a \rightarrow P) \,\square\, (b \rightarrow Q) = (a \rightarrow (P \,\square\, (b \rightarrow Q))) \mid (b \rightarrow ((a \rightarrow P) \,\square\, Q))$

Il est alors possible de calculer la trace d'exécution du système en simplifiant son expression à l'aide des règles énoncées précédemment. On précise d'abord la règle appliquée, puis le résultat de l'application de cette règle.

S_3 $\quad P \square Q = \big((T_0 \mid T_1) \to T_2 \to P\big) \square (T_0 \to T_1 \to Q)$

R_1 $\quad P \square Q = \big((T_0 \to T_2 \to P) \mid (T_1 \to T_2 \to P)\big) \square (T_0 \to T_1 \to Q)$

R_3 $\quad P \square Q = \big((T_0 \to T_2 \to P) \square (T_0 \to T_1 \to Q)\big) \mid \big((T_1 \to T_2 \to P) \square (T_0 \to T_1 \to Q)\big)$

R_5 $\quad P \square Q = \big((T_0 \to T_2 \to P) \square (T_0 \to T_1 \to Q)\big) \mid STOP$

R_2 $\quad P \square Q = (T_0 \to T_2 \to P) \square (T_0 \to T_1 \to Q)$

R_4 $\quad P \square Q = T_0 \to \big((T_2 \to P) \square (T_1 \to Q)\big)$

R_6 $\quad P \square Q = T_0 \to T_2 \to \big(P \square (T_1 \to Q)\big)$

S_1 $\quad P \square Q = T_0 \to T_2 \to \big(\big((T_0 \mid T_1) \to T_2 \to P\big) \square (T_1 \to Q)\big)$

R_1 $\quad P \square Q = T_0 \to T_2 \to \big(\big((T_0 \to T_2 \to P) \mid (T_1 \to T_2 \to P)\big) \square (T_1 \to Q)\big)$

R_3 $\quad P \square Q = T_0 \to T_2 \to \big(\big((T_0 \to T_2 \to P) \square (T_1 \to Q) \mid (T_1 \to T_2 \to P) \square (T_1 \to Q)\big)\big)$

R_5 $\quad P \square Q = T_0 \to T_2 \to \big(\big(STOP \mid (T_1 \to T_2 \to P) \square (T_1 \to Q)\big)\big)$

R_2 $\quad P \square Q = T_0 \to T_2 \to \big((T_1 \to T_2 \to P) \square (T_1 \to Q)\big)$

R_4 $\quad P \square Q = T_0 \to T_2 \to T_1 \to \big((T_2 \to P) \square Q\big)$

R_6 $\quad P \square Q = T_0 \to T_2 \to T_1 \to T_2 \to (P \square Q)$

On obtient ainsi la trace d'exécution $(T_0 T_2 T_1 T_2)^*$.

La Figure 58 représente d'une part, les réseaux de Petri ordinaires de la machine et de son environnement et d'autre part, leur composition parallèle dans le contexte déterministe. Cette composition revient à la fusion des transitions T_0 et T_1 de la machine et de son environnement.

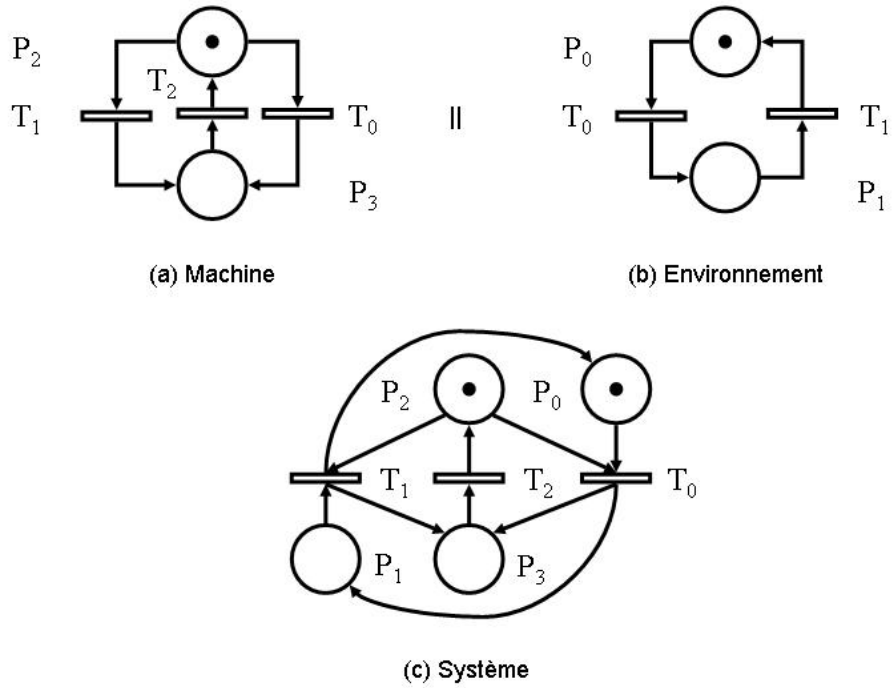


Figure 58 : Composition parallèle déterministe de deux réseaux de Petri ordinaires

A.3 Spécification et composition non déterministe dans CSP

Dans un second temps, on suppose que le système est non déterministe. On a recours à l'opérateur de composition parallèle non déterministe de CSP. Pour simplifier le calcul, on commence par réduire l'expression aux seules actions nécessitant la collaboration des deux processus. Ensuite, on utilise l'opérateur de composition parallèle non déterministe pour générer tous les entrelacements des événements du système. On note arbitrairement $T_0?$ et $T_1?$, les transitions de la machine et $T_0!$ et $T_1!$, les transitions de l'environnement pour les distinguer lors de l'entrelacement. En fait, ces notations correspondent aux notations de CSP pour la communication entre deux processus. T_0 et T_1 définissent alors deux canaux de communication entre P et Q. $T_0!$ définit une écriture et $T_0?$ une lecture sur le canal T_0.

On note $\square$ le choix non déterministe avec influence de l'environnement (c'est l'environnement qui contrôle le choix), $\prod$ le choix non déterministe sans influence de l'environnement (l'environnement n'a aucun contrôle sur le choix) et $|||$ l'opérateur de composition parallèle non déterministe.

Le système s'écrit :

S_1 $\quad P = (T_0? \square T_1?) \rightarrow P$

S_2 $\quad Q = T_0! \rightarrow T_1! \rightarrow Q$

S_3 $\quad P ||| Q = \left((T_0? \square T_1?) \rightarrow P\right) ||| \left(T_0! \rightarrow T_1! \rightarrow Q\right)$

Les règles suivantes sont utilisées pour le calcul.

Le choix non déterministe avec influence de l'environnement dégénère en choix non déterministe sans influence de l'environnement.

$R_1 \qquad (x \to P) \Box (x \to Q) = (x \to P) \prod (x \to Q)$

Le choix non déterministe sans influence de l'environnement est idempotent.

$R_2 \qquad P \prod P = P$

L'opérateur de composition parallèle non déterministe produit l'entrelacement des actions.

$R_3 \qquad (x \to P) \,|||\, (y \to Q) = \left(x \to \left(P \,|||\, (y \to Q)\right)\right) \Box \left(y \to \left((x \to P) \,|||\, Q\right)\right)$

On utilise les règles suivantes, qui ne sont pas directement inscrites dans CSP pour abstraire les communications et réintroduire l'action T_2, ne nécessitant pas la collaboration de P et de Q à la fin du calcul.

$R_4 \qquad (T_0 ? \Box T_1 ?) \to T_0 ! = T_0 ! \to (T_0 ? \Box T_1 ?) = T_0 \to T_2$
$R_5 \qquad (T_0 ? \Box T_1 ?) \to T_1 ! = T_1 ! \to (T_0 ? \Box T_1 ?) = T_1 \to T_2$

Il est alors possible de calculer la trace d'exécution du système en simplifiant son expression à l'aide des règles énoncées précédemment. On précise d'abord la règle appliquée et ensuite le résultat de l'application de cette règle.

$S_3 \qquad P \,|||\, Q = \left((T_0 ? \Box T_1 ?) \to P\right) \,|||\, \left(T_0 ! \to T_1 ! \to Q\right)$

$R_3 \qquad P \,|||\, Q = \left((T_0 ? \Box T_1 ?) \to \left(P \,|||\, \left(T_0 ! \to T_1 ! \to Q\right)\right)\right) \Box \left(T_0 ! \to \left((T_0 ? \Box T_1 ?) \to P \,|||\, \left(T_1 ! \to Q\right)\right)\right)$

$R_3 \qquad P \,|||\, Q = \left((T_0 ? \Box T_1 ?) \to T_0 ! \to \left(P \,|||\, \left(T_1 ! \to Q\right)\right)\right) \Box \left(T_0 ! \to (T_0 ? \Box T_1 ?) \to \left(P \,|||\, \left(T_1 ! \to Q\right)\right)\right)$

$S_1 \qquad P \,|||\, \left(T_1 ! \to Q\right) = \left((T_0 ? \Box T_1 ?) \to P\right) \,|||\, \left(T_1 ! \to Q\right)$

$R_3 \qquad P \,|||\, \left(T_1 ! \to Q\right) = \left((T_0 ? \Box T_1 ?) \to \left(P \,|||\, \left(T_1 ! \to Q\right)\right)\right) \Box \left(T_1 ! \to \left(\left((T_0 ? \Box T_1 ?) \to P\right) \,|||\, Q\right)\right)$

$R_3 \qquad P \,|||\, \left(T_1 ! \to Q\right) = \left((T_0 ? \Box T_1 ?) \to T_1 ! \to \left(P \,|||\, Q\right)\right) \Box \left(T_1 ! \to (T_0 ? \Box T_1 ?) \to \left(P \,|||\, Q\right)\right)$

En réinjectant $P \,|||\, \left(T_1 ! \to Q\right)$, il vient

$$P \,|||\, Q = \left((T_0 ? \Box T_1 ?) \to T_0 ! \to (T_0 ? \Box T_1 ?) \to T_1 ! \to \left(P \,|||\, Q\right)\right)$$
$$\Box \left(T_0 ! \to (T_0 ? \Box T_1 ?) \to T_1 ! \to (T_0 ? \Box T_1 ?) \to \left(P \,|||\, Q\right)\right)$$

Voilà donc le comportement obtenu à l'action interne T_2 près. Le système peut démarrer soit par une action de la machine, soit par une action de l'environnement.

On applique les règles R_4 et R_5 pour d'une part, réintroduire l'action T_2 interne et d'autre part, faire abstraction des communications entre la machine et son environnement.

$R_4, R_5 \qquad P \,|||\, Q = \left(T_0 \to T_2 \to T_1 \to T_2 \to \left(P \,|||\, Q\right)\right) \Box \left(T_0 \to T_2 \to T_1 \to T_2 \to \left(P \,|||\, Q\right)\right)$

$R_1 \qquad P \,|||\, Q = \left(T_0 \to T_2 \to T_1 \to T_2 \to \left(P \,|||\, Q\right)\right) \prod \left(T_0 \to T_2 \to T_1 \to T_2 \to \left(P \,|||\, Q\right)\right)$

R_2 $\quad P|||Q=(T_0\to T_2\to T_1\to T_2)\to(P|||Q)$

On retrouve ainsi la même trace d'exécution $(T_0T_2T_1T_2)^*$ que dans le mode déterministe, au mécanisme de communication entre la machine et son environnement près.

La Figure 59 présente d'une part, les réseaux de Petri ordinaires de la machine et de son environnement et d'autre part, leur composition parallèle dans le contexte non déterministe. Comme il a déjà été mentionné, on a le choix pour démarrer aléatoirement soit par une action de la machine, soit par une action de l'environnement. Sur la Figure 59, c'est l'environnement qui démarre. On peut également constater que l'action interne T_2 s'entrelace naturellement avec les communications entre les deux processus.

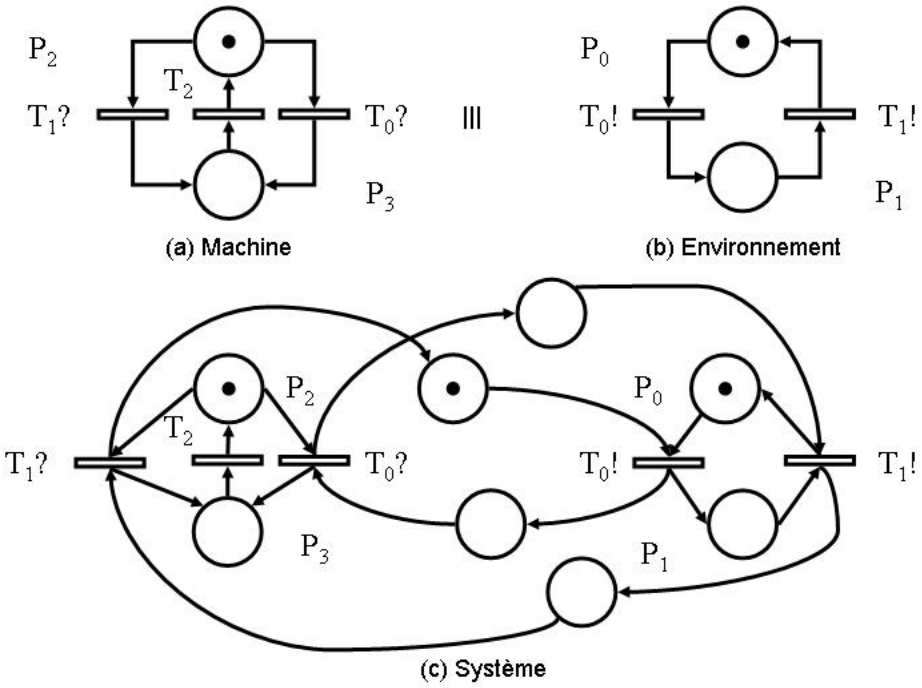


Figure 59 : Composition parallèle non déterministe de deux réseaux de Petri ordinaires

B Annexe du chapitre III

B.1 Modèles de calculs dans SystemC v2.0.1

B.1.1 Modèles de calcul basés sur les canaux de communication élémentaires

Plusieurs canaux de communication élémentaires sont fournis par défaut avec le simulateur et permettent de construire différents types de modèles de calcul. Le canal « sc_signal » représente le signal matériel. Les canaux « sc_mutex », « sc_semaphore » ou « sc_fifo » représentent respectivement un verrou (sémaphore à valeur booléenne), un sémaphore ou une file premier entré, premier sorti (« FIFO » en anglais). Ces derniers peuvent être utilisés dans la modélisation de matériel ou bien la modélisation de logiciel pour l'arbitrage ou la communication à mémoire finie entre différents blocs.

Des modèles de calcul complexes peuvent être implémentés à partir de ces canaux élémentaires [62] : processus séquentiels communicants (« CSP » ou « Communicating Sequential Processes » en anglais), réseaux de Khan (« KPN » ou « Kahn Process Networks » en anglais), systèmes réactifs synchrones (« SR » ou « Synchronous Reactive Systems » en anglais).

B.1.2 Modèles de calcul basés sur les canaux « master/slave »

La conception d'un système sur une puce nécessite la mise au point d'une méthodologie de conception étant donné la complexité du problème à traiter et la variété des modèles en jeu.

SystemC v2.0.1 propose une méthodologie de conception et une bibliothèque de canaux de communication dédiées aux systèmes constitués d'un ou plusieurs processeurs, de processeurs de traitement du signal, de périphériques et de circuits à application spécifique interconnectés par des bus de communication basés sur les ports et les canaux de la bibliothèque maître/esclave comme sur la Figure 60. Cependant, il faut noter que les communications multi point sont considérées comme atomiques et aucun mécanisme d'arbitrage ou de gestion des priorités n'est fourni par défaut.

Cette méthodologie propose de découper la conception en étapes successives du niveau d'abstraction le plus élevé jusqu'au niveau d'abstraction le plus bas. On présente dans la suite les deux niveaux d'abstraction les plus élevés.

Le niveau d'abstraction le plus élevé est le niveau fonctionnel non temporisé (« UTF » ou « Untimed Functionnal » en anglais). Ce niveau permet une modélisation atemporelle de la fonctionnalité et une simulation rapide du système sans faire de choix a priori sur le type d'implémentation : logiciel/matériel ou synchrone/asynchrone. Ses caractéristiques sont les suivantes :

- canaux de communications point à point,
- canaux de communication abstraits,
- modélisation précise des échanges de données et de l'ordre d'exécution,
- exécution des processus en temps nul,
- pas d'hypothèse sur la présence d'une horloge.

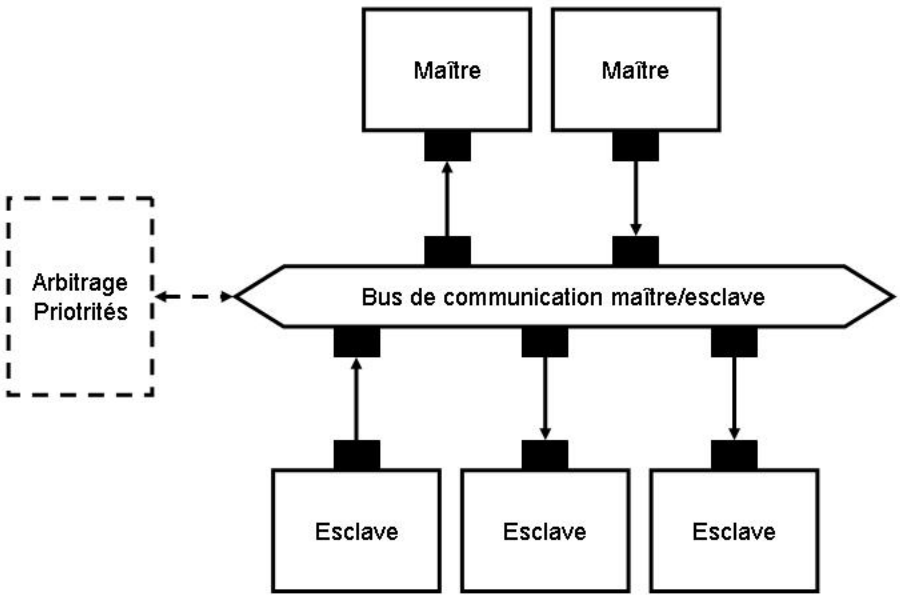


Figure 60 : Architecture d'un système maître/esclave multi point

Le niveau d'abstraction dit fonctionnel temporisé (« TF » ou « Timed Functionnal » en anglais) ajoute d'une part un délai d'exécution aux processus du niveau UTF et raffine la description des canaux de communication. Ce niveau permet une modélisation temporelle de la fonctionnalité dans le but d'estimer les performances du modèle avant de faire des choix d'implémentation.

B.2 Exemple de circuit SystemC v2.0.1 basé sur un canal élémentaire

B.2.1 Oscillateur en anneau

La Figure 61 montre un exemple de code SystemC v2.0.1 pour un oscillateur en anneau. Cet oscillateur est constitué d'une porte « non-et », de délais et de deux inverseurs. Ces éléments sont interconnectés par le canal élémentaire « sc_signal », modélisant le signal matériel. Le signal de mise à zéro est actif à partir de l'état initial, suffisamment longtemps pour permettre le démarrage de l'oscillateur dans un état correct. On a créé un module de délai ad hoc, car le canal élémentaire « sc_signal » ne fournit pas de méthode d'écriture avec retard.

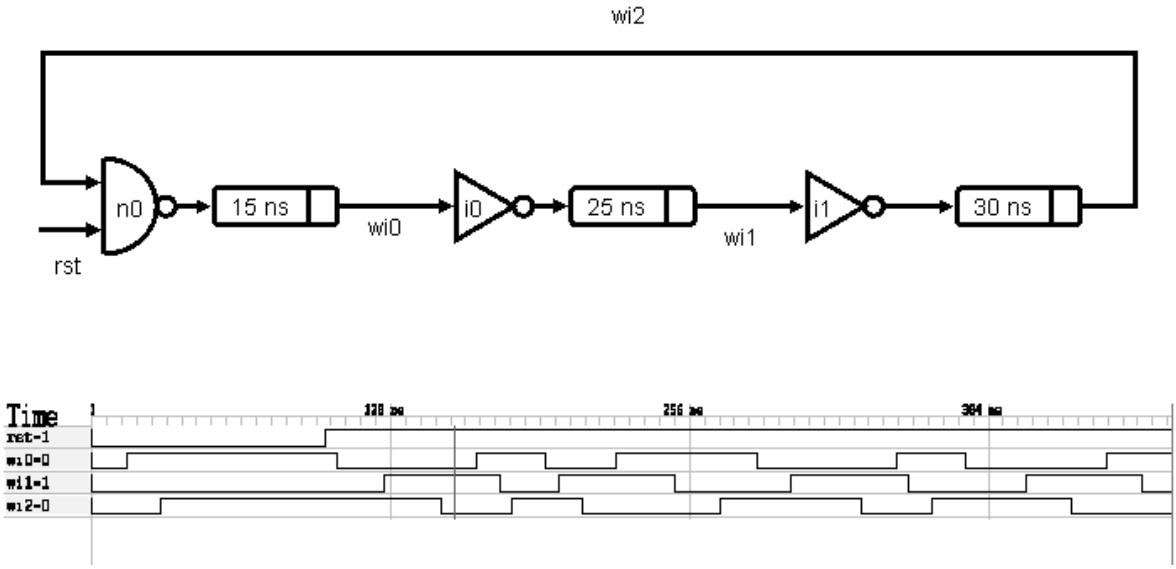


Figure 61 : Schéma électrique de l'oscillateur en anneau et traces d'exécution

B.2.2 Code SystemC v2.0.1 de l'oscillateur en anneau

```
1  #include "SystemC v2.0.1.h"
2
3  SC_MODULE( delay ) {
4
5    sc_in<bool> in;
6    sc_out<bool> out;
7
8    SC_CTOR( delay ) {
9      SC_METHOD( delay_behavior );
10     sensitive << in;
11   }
12
13   void delay_behavior() {
14     out.write( in.read() );
15     next_trigger( get_delay() );
16   }
17
18   void set_delay( sc_time delay ) { del = delay; }
19   sc_time get_delay() { return del; }
20
21   sc_time del;
22
23 };
```

```
24
25 SC_MODULE( inverter ) {
26
27  public:
28    sc_in<bool> in;
29    sc_out<bool> out;
30
31    SC_CTOR( inverter ) {
32      SC_METHOD( inverter_behavior );
33      sensitive << in;
34    }
35
36    void inverter_behavior() {
37      out.write( !in.read() );
38    }
39
40 };
41
42 SC_MODULE( nand ) {
43
44    sc_in<bool>  in0, in1;
45    sc_out<bool> out;
46
47    SC_CTOR( nand ) {
48      SC_METHOD( nand_behavior );
49      sensitive << in0 << in1;
50    }
51
52    void nand_behavior() {
53      out.write( !(in0.read()&in1.read()) );
54    }
55
56 };
57
58 int sc_main(int argc, char** argv) {
59
60    //simulation defaults
61    sc_set_default_time_unit( 1, SC_NS );
62    sc_set_time_resolution( 1, SC_NS );
63
```

```
64  sc_clock rst("reset", sc_time(1000, SC_NS), 0.5, SC_ZERO_TIME,
0);
65  sc_signal<bool> wi0, wi1, wi2;
66  sc_signal<bool> wd0, wd1, wd2;
67
68  nand n0("n0");
69  delay d0("d0");
70  d0.set_delay( sc_time( 15, SC_NS ) );
71
72  inverter i0("inv0");
73  delay d1("d1");
74  d1.set_delay( sc_time( 25, SC_NS ) );
75
76  inverter i1("inv1");
77  delay d2("d2");
78  d2.set_delay( sc_time( 30, SC_NS ) );
79
80  n0.in0(rst);
81  n0.in1(wi2);
82  n0.out(wd0);
83  d0.in(wd0);
84  d0.out(wi0);
85  i0.in(wi0);
86  i0.out(wd1);
87  d1.in(wd1);
88  d1.out(wi1);
89  i1.in(wi1);
90  i1.out(wd2);
91  d2.in(wd2);
92  d2.out(wi2);
93
94  sc_trace_file* tf = sc_create_vcd_trace_file("oscillator");
95  sc_trace(tf, rst,      "rst");
96  sc_trace(tf, wd0,      "wd0");
97  sc_trace(tf, wi0,      "wi0");
98  sc_trace(tf, wd1,      "wd1");
99  sc_trace(tf, wi1,      "wi1");
100     sc_trace(tf, wd2,      "wd2");
101     sc_trace(tf, wi2,      "wi2");
102
103     sc_start( 1500, SC_NS );
```

```
104
105     return 0;
106   }
```

B.3 Illustration des modes d'exécution des co-routines

B.3.1 Porte nand en mode fonctionnel, sensibilité statique et initialisation

```
1  SC_MODULE( nand ) {
2
3    sc_in<bool>  in0, in1;
4    sc_out<bool> out;
5
6    SC_CTOR( nand ) {
7      SC_METHOD( nand_behavior );
8      sensitive << in0 << in1;
9    }
10
11   void nand_behavior() {
12     out.write( !(in0.read()&in1.read()) );
13   }
14 }
```

B.3.2 Porte nand en mode contextuel, sensibilité statique et initialisation

```
1  SC_MODULE( nand ) {
2
3    sc_in<bool>  in0, in1;
4    sc_out<bool> out;
5
6    SC_CTOR( nand ) {
7      SC_THREAD( nand_behavior );
8      sensitive << in0 << in1;
9    }
10
11   void nand_behavior() {
12     out.write( !(in0.read()&in1.read()) );
13   }
14 }
```

B.3.3 Porte nand en mode fonctionnel, sensibilité statique, sans initialisation

```
1  SC_MODULE( nand ) {
2
3    sc_in<bool>  in0, in1;
4    sc_out<bool> out;
5
6    SC_CTOR( nand ) {
7      SC_METHOD( nand_behavior );
8      sensitive << in0 << in1;
9      dont_initialise() ;
10   }
11
12   void nand_behavior() {
13     out.write( !(in0.read()&in1.read()) );
14   }
15 }
```

B.3.4 Porte nand en mode contextuel, sensibilité statique, sans initialisation

```
1  SC_MODULE( nand ) {
2
3    sc_in<bool>  in0, in1;
4    sc_out<bool> out;
5
6    SC_CTOR( nand ) {
7      SC_THREAD( nand_behavior );
8      sensitive << in0 << in1;
9      dont_initialise() ;
10   }
11
12   void nand_behavior() {
13     out.write( !(in0.read()&in1.read()) );
14   }
15 }
```

B.3.5 Porte nand en mode fonctionnel, sensibilité dynamique et initialisation

```
1  SC_MODULE( nand ) {
2
3    sc_in<bool>  in0, in1;
4    sc_out<bool> out;
5
6    SC_CTOR( nand ) {
7      SC_METHOD( nand_behavior );
8    }
9
10   void nand_behavior() {
11     out.write( !(in0.read()&in1.read()) );
12     next_trigger( in0.default_event() | in1.default_event() );
13   }
14 }
```

B.3.6 Porte nand en mode contextuel, sensibilité dynamique et initialisation

```
1  SC_MODULE( nand ) {
2
3    sc_in<bool>  in0, in1;
4    sc_out<bool> out;
5
6    SC_CTOR( nand ) {
7      SC_THREAD( nand_behavior );
8      sensitive << in0 << in1;
9    }
10
11   void nand_behavior() {
12     wait( in0.default_event() | in1.default_event() );
13     out.write( !(in0.read()&in1.read()) );
14   }
15 }
```

B.3.7 Porte nand en mode fonctionnel, sensibilité dynamique, sans initialisation

```
1  SC_MODULE( nand ) {
2
3    sc_in<bool>  in0, in1;
4    sc_out<bool> out;
5
6    SC_CTOR( nand ) {
7      SC_METHOD( nand_behavior );
8      dont_initialise() ;
9    }
10
11   void nand_behavior() {
12     out.write( !(in0.read()&in1.read()) );
13     next_trigger( in0.default_event() | in1.default_event() );
14   }
15 }
```

B.3.8 Porte nand en mode contextuel, sensibilité dynamique, sans initialisation

```
1  SC_MODULE( nand ) {
2
3    sc_in<bool>  in0, in1;
4    sc_out<bool> out;
5
6    SC_CTOR( nand ) {
7      SC_THREAD( nand_behavior );
8      dont_initialise() ;
9    }
10
11   void nand_behavior() {
12     wait( in0.default_event() | in1.default_event() );
13     out.write( !(in0.read()&in1.read()) );
14   }
15 }
```

B.4 Syntaxe SystemC v2.0.1 des notifications d'événement depuis une co-routine

```
1 // Occurrence immédiate d'un événement e
2 e.notify() ;
3
4 // Occurrence d'un événement e à délai nul
5 e.notify( SC_ZERO_TIME ) ;
6
7 // Occurrence d'un événement e retardée d'un délai de 10 ps
8 e.notify( 10, SC_PS) ;
```

B.5 Syntaxe SystemC v2.0.1 des primitives de synchonisation d'une co-routine sur occurrence d'événement

B.5.1 Synchonisation sans minuterie en mode fonctionnel

```
1 // Déclenchement d'une co-routine sur un événement e
2 next_trigger( e );
3 // Déclenchement d'une co-routine sur une conjonction d'événements
4 next_trigger( e0 & e1 );
5 // Déclenchement d'une co-routine sur une disjonction d'événements
6 next_trigger( e0 | e1 );
7 // Déclenchement d'une co-routine après 10 ns
8 next_trigger( sc_time(10, SC_NS) );
```

B.5.2 Synchonisation sans minuterie en mode contextuel

```
1 // Déclenchement d'une co-routine sur un événement e
2 wait( e );
3 // Déclenchement d'une co-routine sur une conjonction d'événements
4 wait( e0 & e1 );
5 // Déclenchement d'une co-routine sur une disjonction d'événements
6 wait( e0 | e1 );
7 // Déclenchement d'une co-routine après 10 ns
8 wait( sc_time(10, SC_NS) );
```

B.5.3 Synchonisation sans minuterie en mode fonctionnel

```
1 // Déclenchement d'une co-routine sur un événement e ou après 10 ns
2 next_trigger( sc_time(10, SC_NS), e );
3 // Déclenchement d'une co-routine sur une conjonction d'événements
4 // ou après 10 ns
5 next_trigger( sc_time(10, SC_NS), e0&e1 );
6 // Déclenchement d'une co-routine sur une disjonction d'événements
7 // ou après 10 ns
8 next_trigger( sc_time(10, SC_NS), e0|e1 );
```

B.5.4 Synchonisation sans minuterie en mode contextuel

```
1 // Déclenchement d'une co-routine sur un événement e ou après 10 ns
2 wait( sc_time(10, SC_NS), e );
3 // Déclenchement d'une co-routine sur une conjonction d'événements
4 // ou après 10 ns
5 wait( sc_time(10, SC_NS), e0&e1 );
6 // Déclenchement d'une co-routine sur une disjonction d'événements
7 // ou après 10 ns
8 wait( sc_time(10, SC_NS), e0|e1 );
```

B.6 Pseudo code du programme principal avec activation du tirage aléatoire des processus

```
1  programme_principal() {
2
3        //
4     // programmation du tirage aléatoire des processus
5     // LCG 48 bits
6        //
7     booléen tirage_aléatoire  = vrai ;
8     entier valeur_initiale    = date_courante ;
9     entier période_lcg_48     = 1 000 000 ;
10
11    tirage_aléatoire_noyau( tirage_aléatoire,
12                            valeur_initiale,
13                            période_lcg_48 ) ;
14
15    //
16       // définition de la résolution temporelle
17       //
18    // description du circuit
```

```
19        //
20    // description des traces
21    //
22
23    simule() ;
24 }
```

C Annexe du chapitre IV

C.1 Production-consommation d'occurrences d'événements

C.1.1 Modèle du protocole production-consommation entre noyau et modèle

```
1 // modification de la classe « événement » du noyau de SystemC
v2.0.1
2 // pour intégrer le drapeau d'occurrence nommé « déclenchement »
3 // la fonction déclenche_processus() réalise la production
4 événement {
5     // crée une occurrence d'événement
6     notifie () ;
7     // déclenche les processus dépendant
8     // de cet événement
9     déclenche_processus () {
10          …
11          // production d'une occurrence dudit événement
12          déclenchement = vrai ;
13    }
14    // drapeau d'occurrence d'événement
15    booléen déclenchement ;
16 }
```

C.1.2 Modèle de structure de choix à deux branches

```
1 // mise en œuvre sur un processus modélisant un choix
2 // à deux branches
3 // le processsus est sensible à deux événements e0 et e1
4 // chaque branche consomme un événement
5 processus {
6     // synchronisation du processus sur une disjonction
7     // d'occurrences d'événements
8     synchronise (e0 ou e1) ;
9     // branche correspondant à l'événement e0
10    si e0.déclenchement = vrai {
11          // calcul dans la branche n°0
12          calcul_0() ;
13          // consommation de l'occurrence
14          // de l'événement e0
15          e0.déclenchement = faux ;
16    }
```

```
17    // branche correspondant à l'événement e1
18    si e1.déclenchement = vrai {
19          // calcul dans la branche n°1
20          calcul_1() ;
21          // consommation de l'occurrence
22          // de l'événement e1
23          e1.déclenchement = faux ;
24    }
25 }
```

C.2 Séparation du calcul et de la communication

C.2.1 Modèle de canal de communication point à point unidirectionnel

```
1  canal {
2     écriture( booléen val ) {
3           valeur = val ;
4     }
5     booléen lecture() {
6           retour valeur ;
7     }
8     événement requête ;
9     booléen valeur ;
10 }
```

C.2.2 Modèle de calcul basé sur le protocole production-consommation

```
1  module {
2     // ports
3     port entrée ;
4     port sortie ;
5     // ancienne valeur
6     booléen var0 ;
7     // processus
8     processus {
9           // variables du processus
10          booléen var1 ;
11          // synchronisation du processus
12          synchronise( entrée.requête ) ;
13          // la production a forcément eu lieu
14          // sinon on ne sera pas dans ce processus
15          si entrée.requête.déclenchement = vrai {
```

```
16                // lecture
17                var0 = entrée.lecture() ;
18                // consommation
19                entrée.requête.déclenchement = faux ;
20          }
21          // calcul (occurrence d'absence d'événement)
22          var1 = non var0 ;
23          // écriture
24          sortie.écriture(var1) ;
25    }
26 }
```

C.3 Canal de communication point à point unidirectionnel direct

```
1  canal {
2      // interface_entrée_canal_sortie_port
3      produit_requête() {
4            requête.notifie() ;
5      }
6      écrit_valeur(booléen valeur) {
7            valeur_entrée = valeur ;
8      }
9      // interface_sortie_canal_entrée_port
10     consomme_requête() {
11           si requête.déclenchement = vrai {
12                met_à_jour_valeur_canal() ;
13                requête.déclenchement = faux ;
14           }
15     }
16     met_à_jour_valeur_canal() {
17           valeur_sortie = valeur_entrée ;
18     }
19     booléen lit_valeur() {
20           retour valeur_sortie ;
21     }
22 }
```

C.4 Canal de communication point à point unidirectionnel à poignée de main

```
1  canal {
2      // interface_entrée_canal_sortie_port
3      produit_requête() {
4            requête.notifie() ;
5      }
6      consomme_acquittement() {
7            si acquittement.déclenchement = vrai
8                  acquittement.déclenchement = faux ;
9      }
10     écrit_valeur(booléen valeur) {
11           valeur_entrée = valeur ;
12     }
13     // interface_sortie_canal_entrée_port
14     produit_acquittement () {
15           acquittement.notifie() ;
16     }
17     consomme_requête() {
18           si requête.déclenchement = vrai {
19                 met_à_jour_valeur_canal() ;
20                 requête.déclenchement = faux ;
21           }
22     }
23     met_à_jour_valeur_sortie_canal() {
24           valeur_sortie = valeur_entrée ;
25     }
26     booléen lit_valeur() {
27           retour valeur_sortie ;
28     }
29 }
```

C.5 Module mono entrée, mono sortie, mono processus

```
1  module {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée ;
4      port <interface_entrée_canal_sortie_port> sortie ;
5      // variables
6      booléen valeur, résultat ;
```

```
7     // processus
8     processus {
9           // synchronisation du processus
10          synchronise( entrée.requête ) ;
11          // lecture
12          entrée.consomme_requête() ;
13          valeur = entrée.lit_valeur();
14          // calcul
15          résultat = calcul(valeur) ;
16          // écriture
17          sortie.produit_requête() ;
18          sortie.écrit_valeur(résultat) ;
19    }
20 }
```

C.6 Module bi entrée, mono sortie, mono processus

```
1  module {
2     // ports
3     port <interface_sortie_canal_entrée_port> entrée_0, entrée_1 ;
4     port <interface_entrée_canal_sortie_port> sortie ;
5     // variables
6     booléen valeur_0, valeur_1, résultat ;
7     // processus
8     processus {
9           // synchronisation du processus sur une disjonction
10          // d'occurrences d'événements
11          synchronise(entrée_0.requête ou entrée_1.requête);
12          // consommation, mise à jour et lecture sur le port 0
13          entrée_0.consomme_requête() ;
14          valeur_0 = entrée_0.lit_valeur();
15          // consommation, mise à jour et lecture sur le port 1
16          entrée_1.consomme_requête() ;
17          valeur_1 = entrée_1.lit_valeur();
18          // calcul
19          résultat = calcul(valeur_0, valeur_1) ;
20          // écriture
21          sortie.produit_requête() ;
22          sortie.écrit_valeur(résultat) ;
23    }
24 }
```

C.7 Module mono entrée, bi sortie, mono processus

```
1 module {
2     // ports
3     port <interface_sortie_canal_entrée_port> entrée ;
4     port <interface_entrée_canal_sortie_port> sortie_0, sortie_1 ;
5     // variables
6     booléen valeur, résultat_0, résultat_1 ;
7     // processus
8     processus {
9           // synchronisation du processus
10          synchronise(entrée.requête);
11          // lecture
12          entrée.consomme_requête() ;
13          valeur = entrée.lit_valeur();
14          // calcul
15          résultat_0 = calcul_0(valeur) ;
16          résultat_1 = calcul_1(valeur) ;
17          // écriture sur la sortie 0
18          sortie_0.produit_requête() ;
19          sortie_0.écrit_valeur(résultat_0) ;
20          // écriture sur la sortie 1
21          sortie_1.produit_requête() ;
22          sortie_1.écrit_valeur(résultat_1) ;
23    }
24 }
```

C.8 Module mono entrée, mono sortie, bi processus

```
1 module {
2     // ports
3     port <interface_sortie_canal_entrée_port> entrée ;
4     port <interface_entrée_canal_sortie_port> sortie ;
5     // compteur de processus initialisé à 0
6     entier compteur ;
7     // variables processus 0
8     booléen valeur_0, résultat_0 ;
9     // processus 0
10    processus_0 {
11          // synchronisation du processus
12          synchronise( entrée.requête ) ;
```

```
13          // lecture
14          entrée.consomme_requête() ;
15          valeur_0 = entrée.lit_valeur();
16          // calcul
17          résultat_0 = calcul() ;
18          // écriture
19          sortie.produit_requête() ;
20          sortie.écrit_valeur(résultat_0) ;
21          // détection de la fin de l'exécution du module
22          compteur = compteur + 1 ;
23          si compteur = 2
24                compteur = 0 ;
25    }
26    // variables processus 1
27    booléen valeur_1, résultat_1 ;
28    // processus 1
29    processus_1 {
30          // synchronisation du processus
31          synchronise( entrée.requête ) ;
32          // lecture
33          entrée.consomme_requête() ;
34          valeur_1 = entrée.lit_valeur();
35          // calcul
36          résultat_1 = calcul() ;
37          // écriture
38          sortie.produit_requête() ;
39          sortie.écrit_valeur(résultat_1) ;
40          // détection de la fin de l'exécution du module
41          compteur = compteur + 1 ;
42          si compteur = 2
43                compteur = 0 ;
44    }
45 }
```

C.9 Exemple de circuit

```
1 circuit {
2     module m0, m1 ;
3     canal c0, c1 ;
4     m0.entrée(c1) ;
5     m0.sortie(c0) ;
6     m1.entrée(c0) ;
7     m1.sortie(c1) ;
8 }
```

C.10 Consommateur

```
1 consommateur {
2     // ports
3     port <interface_sortie_canal_entrée_port> entrée ;
4     // variables
5     booléen valeur ;
6     // processus
7     processus {
8           // synchronisation du processus
9           synchronise( entrée.requête ) ;
10          // lecture
11          entrée.consomme_requête() ;
12          valeur = entrée.lit_valeur();
13          // calcul
14          calcul(valeur) ;
15    }
16}
```

C.11 Oscillateur de Muller

C.11.1 Producteur

C.11.1.1 Générateur

```
1 générateur {
2     // ports
3     port <interface_entrée_canal_sortie_port> sortie ;
4     // variables
5     booléen résultat ;
6     // processus
```

```
7     processus {
8           // émission d'un 1
9           résultat = vrai ;
10          si résultat != sortie.valeur_entrée {
11                // écriture
12                sortie.produit_requête() ;
13                sortie.écrit_valeur(résultat) ;
14          }
15          sinon
16                détecte_transition_ineffective() ;
17          // synchronisation du processus
18          // sur un acquittement
19          synchronise( sortie.acquittement ) ;
20          sortie.consomme_acquittement() ;
21          // émission d'un 0
22          résultat = faux ;
23          si résultat != sortie.valeur_entrée {
24                // écriture
25                sortie.produit_requête() ;
26                sortie.écrit_valeur(résultat) ;
27          }
28          sinon
29                détecte_transition_ineffective() ;
30          // synchronisation du processus
31          // sur un acquittement
32          synchronise( sortie.acquittement ) ;
33          sortie.consomme_acquittement() ;
34          // fin d'exécution du processus
35    }
36 }
```

C.11.1.2 Adaptateur

```
1  adaptateur {
2     // ports
3     port <interface_sortie_canal_entrée_port> entrée ;
4     port <interface_entrée_canal_sortie_port> sortie ;
5     // variables
6     booléen valeur, résultat ;
7     // processus
8     processus {
```

```
9           // synchronisation du processus
10          synchronise( entrée.requête ) ;
11          // lecture
12          entrée.consomme_requête() ;
13          valeur = entrée.lit_valeur();
14          entrée.produit_acquittement() ;
15          // calcul
16          résultat = valeur ;
17          si résultat != sortie.valeur_entrée {
18                // écriture
19                sortie.produit_requête() ;
20                sortie.écrit_valeur(résultat) ;
21          }
22          sinon
23                détecte_transition_ineffective() ;
24    }
25 }
```

C.11.1.3 Circuit du producteur

```
1  producteur {
2      // ports
3      port <interface_entrée_canal_sortie_port> sortie ;
4      générateur g ;
5      adaptateur a ;
6      canal c ;
7      g.sortie(c) ;
8      a.entrée(c) ;
9      a.sortie(sortie) ;
10 }
```

C.11.2 Fourche

```
1  fourche {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée ;
4      port <interface_entrée_canal_sortie_port> sortie_0, sortie_1 ;
5      // compteur de processus initialisé à 0
6      entier compteur ;
7      // variables du processus 0
8      booléen valeur_0, garde_0 ;
9      // processus 0
```

```
10    processus_0 {
11          // synchronisation du processus 0
12          synchronise( entrée.requête ) ;
13          // lecture
14          entrée.consomme_requête() ;
15          valeur_0 = entrée.lit_valeur();
16          // calcul
17          garde_0 = valeur_0 ;
18          si garde_0 = vrai {
19                // écriture sur la sortie n°0
20                si sortie_0.valeur_entrée != faux {
21                      sortie_0.produit_requête() ;
22                      sortie_0.écrit_valeur(faux) ;
23                }
24                sinon
25                      détecte_transition_ineffective() ;
26                // écriture sur la sortie n°1
27                si sortie_1.valeur_entrée != faux {
28                      sortie_1.produit_requête() ;
29                      sortie_1.écrit_valeur(faux) ;
30                }
31                sinon
32                      détecte_transition_ineffective() ;
33          }
34          // détection de la fin de l'exécution du module
35          compteur = compteur + 1 ;
36          si compteur = 2
37                compteur = 0 ;
38    }
39    // variables du processus 1
40    booléen valeur_1, garde_1 ;
41    // processus 1
42    processus_1 {
43          // synchronisation du processus 1
44          synchronise( entrée.requête ) ;
45          // lecture
46          entrée.consomme_requête() ;
47          valeur_1 = entrée.lit_valeur();
48          // calcul
49          garde_1 = valeur_1 ;
```

```
50          si garde_1 = vrai {
51                // écriture sur la sortie n°0
52                si sortie_0.valeur_entrée != vrai {
53                      sortie_0.produit_requête() ;
54                      sortie_0.écrit_valeur(vrai) ;
55                }
56                sinon
57                      détecte_transition_ineffective() ;
58                // écriture sur la sortie n°1
59                si sortie_1.valeur_entrée != vrai {
60                      sortie_1.produit_requête() ;
61                      sortie_1.écrit_valeur(vrai) ;
62                }
63                sinon
64                      détecte_transition_ineffective() ;
65          }
66          // détection de la fin de l'exécution du module
67          compteur = compteur + 1 ;
68          si compteur = 2
69                compteur = 0 ;
70    }
71 }
```

C.11.3 Porte « non-ou »

```
1  non_ou {
2     // ports
3     port <interface_sortie_canal_entrée_port> entrée_0, entrée_1 ;
4     port <interface_entrée_canal_sortie_port> sortie ;
5     // compteur de processus initialisé à 0
6     entier compteur ;
7     // variables du processus 0
8     booléen valeur_0_0, valeur_1_0, garde_0 ;
9     // processus 0
10    processus_0 {
11          // synchronisation du processus 0
12          synchronise( entrée_0.requête ou entrée_1.requête ) ;
13          // lecture
14          entrée_0.consomme_requête() ;
15          valeur_0_0 = entrée.lit_valeur();
16          entrée_1.consomme_requête() ;
```

```
17          valeur_1_0 = entrée.lit_valeur();
18          // calcul
19          garde_0 = valeur_0_0 ou valeur_1_0 ;
20          si garde_0 = vrai {
21                // écriture
22                si sortie.valeur_entrée != faux {
23                      sortie.produit_requête() ;
24                      sortie.écrit_valeur(faux) ;
25                }
26                sinon
27                      détecte_transition_ineffective() ;
28          }
29          // détection de la fin de l'exécution du module
30          compteur = compteur + 1 ;
31          si compteur = 2
32                compteur = 0 ;
33    }
34    // variables du processus 1
35    booléen valeur_1_1, valeur_0_1, garde_1 ;
36    // processus 1
37    processus_1 {
38          // synchronisation du processus 1
39          synchronise( entrée_0.requête ou entrée_1.requête ) ;
40          // lecture
41          entrée_0.consomme_requête() ;
42          valeur_0_1 = entrée.lit_valeur();
43          entrée_1.consomme_requête() ;
44          valeur_1_1 = entrée.lit_valeur();
45          // calcul
46          garde_1 = non valeur_0_1 et non valeur_1_1 ;
47          si garde_1 = vrai {
48                // écriture
49                si sortie.valeur_entrée != vrai {
50                      sortie.produit_requête() ;
51                      sortie.écrit_valeur(vrai) ;
52                }
53                sinon
54                      détecte_transition_ineffective() ;
55          }
56          // détection de la fin de l'exécution du module
```

```
57          compteur = compteur + 1 ;
58          si compteur = 2
59                compteur = 0 ;
60    }
61 }
```

C.11.4 Porte de Muller

```
1  muller {
2     // ports
3     port <interface_sortie_canal_entrée_port> entrée_0, entrée_1 ;
4     port <interface_entrée_canal_sortie_port> sortie ;
5     // compteur de processus initialisé à 0
6     entier compteur ;
7     // variables du processus 0
8     booléen valeur_0_0, valeur_1_0, garde_0 ;
9     // processus 0
10    processus_0 {
11          // synchronisation du processus 0
12          synchronise( entrée_0.requête ou entrée_1.requête ) ;
13          // lecture
14          entrée_0.consomme_requête() ;
15          valeur_0_0 = entrée.lit_valeur();
16          entrée_1.consomme_requête() ;
17          valeur_1_0 = entrée.lit_valeur();
18          // calcul
19          garde_0 = non valeur_0_0 et non valeur_1_0 ;
20          si garde_0 = vrai {
21                // écriture
22                si sortie.valeur_entrée != faux {
23                      sortie.produit_requête() ;
24                      sortie.écrit_valeur(faux) ;
25                }
26                sinon
27                      détecte_transition_ineffective() ;
28          }
29          // détection de la fin de l'exécution du module
30          compteur = compteur + 1 ;
31          si compteur = 2
32                compteur = 0 ;
33    }
```

```
34    // variables du processus 1
35    booléen valeur_1_1, valeur_0_1, garde_1 ;
36    // processus 1
37    processus_1 {
38          // synchronisation du processus 1
39          synchronise( entrée_0.requête ou entrée_1.requête ) ;
40          // lecture
41          entrée_0.consomme_requête() ;
42          valeur_0_1 = entrée.lit_valeur();
43          entrée_1.consomme_requête() ;
44          valeur_1_1 = entrée.lit_valeur();
45          // calcul
46          garde_1 = valeur_0_1 et valeur_1_1 ;
47          si garde_1 = vrai {
48                // écriture
49                si sortie.valeur_entrée != vrai {
50                      sortie.produit_requête() ;
51                      sortie.écrit_valeur(vrai) ;
52                }
53                sinon
54                      détecte_transition_ineffective() ;
55          }
56          // détection de la fin de l'exécution du module
57          compteur = compteur + 1 ;
58          si compteur = 2
59                compteur = 0 ;
60    }
61 }
```

C.11.5 Fil

```
1  fil {
2     // ports
3     port <interface_sortie_canal_entrée_port> entrée ;
4     port <interface_entrée_canal_sortie_port> sortie ;
5     // compteur de processus initialisé à 0
6     entier compteur ;
7     // variables du processus 0
8     booléen valeur_0, garde_0 ;
9     // processus 0
10    processus_0 {
```

```
11          // synchronisation du processus 0
12          synchronise( entrée.requête ) ;
13          // lecture
14          entrée.consomme_requête() ;
15          valeur_0 = entrée.lit_valeur();
16          // calcul
17          garde_0 = valeur_0 ;
18          si garde_0 = vrai {
19                // écriture sur la sortie
20                si sortie.valeur_entrée != faux {
21                      sortie.produit_requête() ;
22                      sortie.écrit_valeur(faux) ;
23                }
24                sinon
25                      détecte_transition_ineffective() ;
26          }
27          // détection de la fin de l'exécution du module
28          compteur = compteur + 1 ;
29          si compteur = 2
30                compteur = 0 ;
31    }
32    // variables du processus 1
33    booléen valeur_1, garde_1 ;
34    // processus 1
35    processus_1 {
36          // synchronisation du processus 1
37          synchronise( entrée.requête ) ;
38          // lecture
39          entrée.consomme_requête() ;
40          valeur_1 = entrée.lit_valeur();
41          // calcul
42          garde_1 = valeur_1 ;
43          si garde_1 = vrai {
44                // écriture sur la sortie
45                si sortie.valeur_entrée != vrai {
46                      sortie.produit_requête() ;
47                      sortie.écrit_valeur(vrai) ;
48                }
49                sinon
50                      détecte_transition_ineffective() ;
```

```
51          }
52          // dét ection de la fin de l'exécution du module
53          compteur = compteur + 1 ;
54          si compteur = 2
55                compteur = 0 ;
56    }
57 }
```

C.11.6 Circuit de l'oscillateur de Muller

```
1  oscillateur_muller {
2      // portes
3      producteur raz ;
4      fourche fraz, t1 ;
5      nor p0, p1 ;
6      muller t0 ;
7      fil p2 ;
8      // canaux
9      canal raz_fraz, fraz_p0, fraz_p1 ;
10     canal p2_t1, t1_p0, t1_p1, p0_t0, p1_t0, t0_p2 ;
11     // producteur
12     raz.sortie(raz_fraz) ;
13     // fourche
14     fraz.entrée(raz_fraz) ;
15     fraz.sortie_0(fraz_p0) ;
16     fraz.sortie_0(fraz_p1) ;
17     // fourche
18     t1.entrée(p2_t1) ;
19     t1.sortie_0(t1_p0) ;
20     t1.sortie_1(t1_p1) ;
21     // nor
22     p0.entrée_0(fraz_p0) ;
23     p0.entrée_1(t1_p0) ;
24     p0.sortie(p0_t0) ;
25     // nor
26     p1.entrée_0(fraz_p1) ;
27     p1.entrée_1(t1_p1) ;
28     p1.sortie(p1_t0) ;
29     // muller
30     t0.entrée_0(p0_t0) ;
31     t0.entrée_1(p1_t0) ;
```

```
32    t0.sortie(t0_p2) ;
33    // fil
34    p2.entrée(t0_p2) ;
35    p2.sortie(p2_t1) ;
36 }
```

C.11.7 Temporisation possible de l'oscillateur de Muller

```
1  oscillateur_muller {
2     // circuit
3     // temporisation
4     raz.délai(250 ps) ;
5     raz_fraz.délai(0 ps) ;
6     fraz_p0.délai(100 ps) ;
7     fraz_p1.délai(200 ps) ;
8     p0_t0.délai(50 ps) ;
9     p1_t0.délai(150 ps) ;
10    t0_p2.délai(10 ps) ;
11    p2_t1.délai(20 ps) ;
12    t1_p0.délai(30 ps) ;
13    t1_p1.délai(40 ps) ;
14 }
```

D Annexe du chapitre V

D.1 Canal de communication point à point unidirectionnel avec prise en compte de la persistance

```
1  canal {
2      // interface_entrée_canal_sortie_port
3      verrouille_canal() {
4            verrou = vrai ;
5      }
6      produit_requête() {
7            si verrou = faux {
8                  verrouille_canal() ;
9            }
10           sinon {
11                 détecte_erreur_persistance() ;
12                 arrête_simulation() ;
13           }
14           requête.notifie() ;
15     }
16     …
17     // interface_sortie_canal_entrée_port
18     déverrouille_canal() {
19           verrou = faux ;
20     }
21     consomme_requête() {
22           si requête.déclenchement = vrai {
23                 si verrou = vrai {
24                       déverrouille_canal() ;
25                 }
26                 met_à_jour_valeur_canal() ;
27                 requête.déclenchement = faux ;
28           }
29     }
30     …
31 }
```

D.2 Modèle SystemC v2.0.1 d'une porte mono entrée, mono sortie avec prise en compte de la propriété de sûreté

```
1  module {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée ;
4      port <interface_entrée_canal_sortie_port> sortie ;
5      // compteur de processus initialisé à 0
6      entier compteur ;
7      // variables du processus 0
8      booléen valeur_0, garde_0 ;
9      // processus 0
10     processus_0 {
11           // synchronisation du processus 0
12           synchronise( entrée.requête ) ;
13           // lecture
14           entrée.consomme_requête() ;
15           valeur_0 = entrée.lit_valeur();
16           // calcul
17           garde_0 = calcul(valeur_0) ;
18           si garde_0 = vrai et garde_1 = vrai {
19                 détecte_erreur_sûreté() ;
20                 arrête_simulation() ;
21           }
22           si garde_0 = vrai {
23                 // écriture
24                 si sortie.valeur_entrée != faux {
25                       sortie.produit_requête() ;
26                       sortie.écrit_valeur(faux) ;
27                 }
28                 sinon
29                       détecte_transition_ineffective() ;
30           }
31           // détection de la fin de l'exécution du module
32           // et réinitialisation des gardes
33           compteur = compteur + 1 ;
34           si compteur = 2 {
35                 garde_0 = 0 ;
36                 garde_1 = 0 ;
37                 compteur = 0 ;
```

```
38          }
39    }
40    // variables du processus 1
41    booléen valeur_1, garde_1 ;
42    // processus 1
43    processus_1 {
44          // synchronisation du processus 1
45          synchronise( entrée.requête ) ;
46          // lecture
47          entrée.consomme_requête() ;
48          valeur_1 = entrée.lit_valeur();
49          // calcul
50          garde_1 = calcul(valeur_1) ;
51          si garde_0 = vrai et garde_1 = vrai {
52                detecte_erreur_sûreté() ;
53                arrête_simulation() ;
54          }
55          si garde_1 = vrai {
56                // écriture
57                si sortie.valeur_entrée != vrai {
58                      sortie.produit_requête() ;
59                      sortie.écrit_valeur(vrai) ;
60                }
61                sinon
62                      detecte_transition_ineffective() ;
63          }
64          // détection de la fin de l'exécution du module
65          // et réinitialisation des gardes
66          compteur = compteur + 1 ;
67          si compteur = 2 {
68                garde_0 = 0 ;
69                garde_1 = 0 ;
70                compteur = 0 ;
71          }
72    }
73 }
```

D.3 Instrumentation du modèle pour la vérification de la propriété de vivacité

D.3.1 Modèle SystemC v2.0.1 du producteur avec prise en compte de la vivacité

D.3.2 Générateur

```
1  générateur {
2      // ports
3      port <interface_entrée_canal_sortie_port> sortie ;
4      // variables
5      booléen résultat ;
6      // processus
7      processus {
8            // calcul
9            résultat = calcul() ;
10           si résultat != sortie.valeur_entrée {
11                 // écriture
12                 sortie.produit_requête() ;
13                 sortie.écrit_valeur(résultat) ;
14           }
15           sinon
16                 détecte_transition_ineffective() ;
17           // synchronisation avec minuterie du processus
18           // sur un acquittement
19           synchronise( instant_blocage, sortie.acquittement ) ;
20           sortie.consomme_acquittement() ;
21     }
22 }
```

D.3.3 Adaptateur

```
1  adaptateur {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée ;
4      port <interface_entrée_canal_sortie_port> sortie ;
5      // variables
6      booléen valeur, résultat ;
7      // processus
8      processus {
9            // synchronisation avec minuterie du processus
10           synchronise( instant_blocage, entrée.requête ) ;
```

```
11          // lecture
12          entrée.consomme_requête() ;
13          valeur = entrée.lit_valeur();
14          entrée.produit_acquittement() ;
15          // calcul
16          résultat = valeur ;
17          si résultat != sortie.valeur_entrée {
18                // écriture
19                sortie.produit_requête() ;
20                sortie.écrit_valeur(résultat) ;
21          }
22          sinon
23                détecte_transition_ineffective() ;
24    }
25 }
```

D.3.4 Programme principal et activation de la détection des blocages

```
1  programme_principal() {
2      // programmation de la détection des blocages
3      booléen blocage        = vrai ;
4      …
5      si blocage = vrai
6            simule(instant_infini) ;
7      sinon
8            simule() ;
9  }
```

D.4 Générateur de jetons

D.4.1 Producteur sur événement

D.4.2 Générateur sur événement

```
1  générateur_ev {
2      // ports
3      port <interface_entrée_canal_sortie_port> sortie ;
4      // processus
5      processus {
6            // production d'une requête
7            sortie.produit_requête() ;
8            // synchronisation du processus
```

```
9           // sur un acquittement
10          synchronise( sortie.acquittement ) ;
11          sortie.consomme_acquittement() ;
12          // fin d'exécution du processus
13    }
14 }
```

D.4.3 Adaptateur sur événement

```
1  adaptateur_ev {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée ;
4      port <interface_entrée_canal_sortie_port> sortie ;
5      // processus
6      processus {
7            // synchronisation du processus
8            synchronise( entrée.requête ) ;
9            // consommation
10           entrée.consomme_requête() ;
11           entrée.produit_acquittement() ;
12           // production
13           sortie.produit_requête() ;
14     }
15 }
```

D.4.4 Circuit du producteur sur événement

```
1  producteur_ev {
2      // ports
3      port <interface_entrée_canal_sortie_port> sortie ;
4      générateur_ev g ;
5      adaptateur_ev a ;
6      canal c ;
7      g.sortie(c) ;
8      a.entrée(c) ;
9      a.sortie(sortie) ;
10 }
```

D.4.5 Porte « fourche » sur événement

```
1  fourche_ev {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée ;
4      port <interface_entrée_canal_sortie_port> sortie_0, sortie_1 ;
5      // processus
6      processus {
7            // synchronisation du processus
8            synchronise( entrée.requête ) ;
9            // consommation
10           entrée.consomme_requête() ;
11           // production sur la sortie n°0
12           sortie_0.produit_requête() ;
13           // production sur la sortie n°1
14           sortie_1.produit_requête() ;
15     }
16 }
```

D.4.6 Porte « ou » sur événement

```
1  ou_ev {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée_0, entrée_1 ;
4      port <interface_entrée_canal_sortie_port> sortie ;
5      // processus
6      processus {
7            // synchronisation du processus
8            synchronise( entrée_0.requête ou entrée_1.requête ) ;
9            // consommation
10           entrée_0.consomme_requête() ;
11           entrée_1.consomme_requête() ;
12           // production
13           sortie.produit_requête() ;
14     }
15 }
```

D.4.7 Porte « fil » sur événement

```
1  fil_ev {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée ;
4      port <interface_entrée_canal_sortie_port> sortie ;
5      // processus
6      processus {
7            // synchronisation du processus
8            synchronise( entrée.requête ) ;
9            // consommation
10           entrée.consomme_requête() ;
11           // production
12           sortie.produit_requête() ;
13     }
14 }
```

D.4.8 Porte « ou exclusif » sur événement

```
1  ou_ex_ev {
2      // ports
3      port <interface_sortie_canal_entrée_port> entrée_0, entrée_1 ;
4      port <interface_entrée_canal_sortie_port> sortie ;
5      // compteur de processus initialisé à 0
6      entier compteur ;
7      // variables du processus 0
8      booléen garde_0 ;
9      // processus 0
10     processus_0 {
11           // synchronisation du processus 0
12           synchronise( entrée_0.requête ou entrée_1.requête ) ;
13           // détecte l'origine de l'activation du processus
14           si entrée_0.requête.déclenchement = vrai
15                 garde_0 = vrai ;
16           // consommation
17           entrée_0.consomme_requête() ;
18           // détecte une erreur de sûreté
19           si garde_0 = vrai et garde_1 = vrai {
20                 détecte_erreur_sûreté() ;
21                 arrête_simulation() ;
22           }
23           // production
```

```
24          si garde_0 = vrai {
25                // production
26                sortie.produit_requête() ;
27                }
28          sinon
29                détecte_transition_ineffective() ;
30          // détection de la fin de l'exécution du module
31          // et réinitialisation des gardes
32          compteur = compteur + 1 ;
33          si compteur = 2 {
34                garde_0 = 0 ;
35                garde_1 = 1 ;
36                compteur = 0 ;
37          }
38    }
39    // variables du processus 1
40    booléen garde_1 ;
41    // processus 1
42    processus_1 {
43          // synchronisation du processus 1
44          synchronise( entrée_0.requête ou entrée_1.requête ) ;
45          // détecte l'origine de l'activation du processus
46          si entrée_1.requête.déclenchement = vrai
47                garde_1 = vrai ;
48          // consommation
49          entrée_1.consomme_requête() ;
50          // détecte une erreur de sûreté
51          si garde_0 = vrai et garde_1 = vrai {
52                détecte_erreur_sûreté() ;
53                arrête_simulation() ;
54          }
55          // production
56          si garde_1 = vrai {
57                // production
58                sortie.produit_requête() ;
59                }
60          sinon
61                détecte_transition_ineffective() ;
62          // détection de la fin de l'exécution du module
63          // et réinitialisation des gardes
```

```
64          compteur = compteur + 1 ;
65          si compteur = 2 {
66                garde_0 = 0 ;
67                garde_1 = 0 ;
68                compteur = 0 ;
69          }
70    }
71 }
```

D.4.9 Circuit du générateur de jetons

```
1  générateur_jetons {
2     // portes
3     producteur_ev init ;
4     ou_ev p0 ;
5     fourche_ev t0 ;
6     ou_ex_ev p1 ;
7     fil_ev t1 ;
8     // canaux
9     canal init_p0, p0_t0, t0_p10, t0_p11, p1_t1, t1_p0 ;
10    // producteur_ev
11    init.sortie(init_p0) ;
12    // ou_ev
13    p0.entrée_0(init_p0) ;
14    p0.entrée_1(t1_p0) ;
15    p0.sortie(p0_t0) ;
16    // fourche_ev
17    t0.entrée(p0_t0) ;
18    t0.sortie_0(t0_p10) ;
19    t0.sortie_1(t0_p11) ;
20    // ou_ex_ev
21    p1.entrée_0(t0_p10) ;
22    p1.entrée_1(t0_p11) ;
23    p1.sortie(p1_t1) ;
24    // fil_ev
25    t1.entrée(p1_t1) ;
26    t1.sortie(t1_p0) ;
27 }
```

Résumé

Suivant les recommandations de l'ITRS 2003, il convient de s'intéresser aux circuits numériques asynchrones pour préparer l'avenir de la conception des circuits numériques. Sur le plan théorique, nous établissons le théorème de l'insensibilité aux délais des circuits numériques asynchrones. Pour spécifier un circuit numérique asynchrone par un programme faisant abstraction des délais, il faut et il suffit que le circuit vérifie trois propriétés fondamentales, que nous appelons propriétés d'insensibilité aux délais, i.e. persistance, sûreté et vivacité. Sur le plan pratique, nous choisissons le standard SystemC v2.0.1 de conception de systèmes numériques pour élaborer le premier modèle de circuits numériques asynchrones instrumenté pour la vérification des propriétés d'insensibilité aux délais, intégré dans un standard de conception de systèmes numériques. Nous mettons ce modèle en œuvre sur des exemples de taille réduite, mais significatifs, avant d'élargir la perspective.

Mots Clés

Modélisation, Simulation, Vérification, Circuits numériques asynchrones, standard SystemC v2.0.1

Title

Modeling, simulating and verifying asynchronous digital circuits in SystemC v2.0.1 standard

Abstract

ITRS 2003 forecasts importance of asynchronous digital design style growth with digital system complexity increase and technology shrinking. From a theoretical perspective, discrete event systems encompass asynchronous digital circuits. A necessary and sufficient condition to abstract delays from an asynchronous digital circuit specification is the guarantee of delay insensitive properties, namely persistence, safety and liveness. From a practical perspective, SystemC v2.0.1 digital system design standard is targeted and adapted to support asynchronous digital circuits modeling and simulation, including delay insensitive properties verification. The model is validated on small but significant examples. Finally, perspective opens out. First, a finer theoretical study should be lead. Second, SystemC v2.0.1 standard should evolve to integrate asynchronous discrete event systems semantics. Third, the model of asynchronous digital circuits based on SystemC v2.0.1 should be improved. At last, asynchronous digital design flow based on SystemC v2.0.1 should be completed with formal verification, circuit synthesis and test.

Keywords

Modeling, simulation, verification, asynchronous digital circuits, SystemC v2.0.1 standard

Intitulé et adresse du laboratoire

Laboratoire TIMA
Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs
46, avenue Félix Viallet
38031 GRENOBLE Cedex
France

ISBN : 2-84813-029-6

ISBNE : 2-84813-030-X