A stepwise compositional approach to model and analyze system C designs at the transactional level and the delta cycle level

Nesrine Harrath

le cnam

ÉCOLE DOCTORALE INFORMATIQUE, TÉLÉCOMMUNICATIONS ET ÉLECTRONIQUE de Paris (EDITE)

ÉQUIPE VESPA - LABORATOIRE CEDRIC

LABORATOIRE U2IS - ENSTA ParisTech

THESE DE DOCTORAT

présentée par : **Nesrine HARRATH**

soutenue le : **04 Novembre 2014**

pour obtenir le grade de : **Docteur du Conservatoire National des Arts et Métiers**

Discipline/Spécialité : **Informatique**

A Stepwise Compositional Approach to Model and Analyze SystemC Designs at the Transactional Level and the Delta Cycle Level

THÈSE DIRIGÉE PAR

Prof. Kamel Barkaoui	Département Info, Cedric, Cnam Paris.
Prof. Bruno Monsuez	ENSTA ParisTech-U2IS.

RAPPORTEURS

Prof. Alain Merigot	IEF, Université Paris-Sud.
Prof. Mohamed Shawky	Université de Technologie de Compiègne, Heudiasyc Lab.

EXAMINATEURS

Dr. Samia Bouzefrane	Cedric, Cnam	Examinatrice
Prof. Zhiwu Li	System Control Automation Group, Xidian University	Examinateur
Prof. Pierre Paradinas	Cedric, Cnam	Examinateur
Dr. Franck Vedrine	Institut Carnot, CEA LIST	Examinateur

Remerciements

Passage obligatoire mais sincère envers les personnes qui ont contribué de loin ou de prêt á la réussite de ce travail.

Je remercie sincèrement Mohamed Shawky et Alain Merigot pour avoir rapporté cette thèse. Je suis extrêmement reconnaissante pour leur travail et surtout pour leurs commentaires avisés et pertinents. Je suis aussi très honorée de compter comme membre de jury Pierre Paradinas, Samia Bouzefrane, Franck Vedrine et Zhiwu Li.

Je suis très réconnaissante envers mon co-directeur de thèse Bruno Monsuez avec qui j'ai travaillé pendant mon stage de master en 2009 et puis ma thèse. J'ai pris énormément de plaisir d'être une de ses doctorantes. Malgré ses engagements, il a su m'écouter, discuter avec moi et surtout avoir confiance en mes capacités.

J'ai une grande gratitude envers mon directeur de thèse Kamel Barkaoui qui m'a accueillie au sein de son équipe VESPA/CEDRIC au CNAM. Il m'a beaucoup aidée que ce soit à l'échelle professionnelle qu à l'échelle humaine. Je lui remercie aussi pour nos discussions sincères.

Je n'oublie pas tous mes collègues au laboratoire Informatique et Ingénierie des Systèmes à l'ENSTA ParisTech.

Un grand merci à tous mes collègues du travail à la RATP avec qui j'ai passé une agréable année, c'est grâce à leur soutien et bonne humeur que j'ai pu accomplir aisément et tranquillement ma dernière année de thèse : Khalid, Sebti, Nassim, Elena, Marie-Hélène.

Milles merci à ma chère Nayma, avec qui j'ai passé des moments de folie qui m'ont permise d'oublier mes moments de stress. *Muchas gracias.*

Un grand merci à ma très chère copine Nesrine et ses deux filles Alaa et Oumayma. A mes copines : Natalia, Catherine, Abir, Khawla, Islem, Souhair et Ouided. A Mondher, je te remercie, pour avoir supporté mes hauts et mes bas.

Pour finir, je tiens à remercier tous les membres de ma famille qui m'ont soutenue et supportée pendant ces trois ans...et qui le font depuis bien plus longtemps ! A mes nièces et neveux : Wissal, Rabeb, Chahd, Ayoub, Adem, Mohamed, Tasnim, Dhia, Loujayn et Mohamed.

Je remercie particulièrement ma très chère maman Meriem qui y tient une place très importante dans ma vie.

Nesrine Harrath

Je dédie ce travail à ma mère Meriem,

à la mémoire de mon père Mohamed

et à la Tunisie, mon pays que j'aime.

Résumé Détaillé

La conception des systèmes embarqués est une conception à la fois matérielle et logicielle. Traditionnellement, les composants logiciels d'un système sont écrits dans un langage de programmation comme C ou C++, alors que la partie matérielle est écrite dans un langage de description tels que VHDL ou Verilog. Cette approche a plusieurs inconvénients : tout d'abord, le concepteur était obligé d'apprendre et de comprendre plusieurs langages de programmation. De plus, au début du processus de la conception , il est souvent difficile de savoir laquelle des parties est à implémenter dans le matériel ou le logiciel. En outre, si la partition de la conception du matériel et du logiciel doit être modifiée plus tard, des coûts et des délais de conception s'ajoutent. Cela motive l'idée d'utiliser des langages uniformes de conception des systèmes afin de fournir la clarté, l'exhaustivité et l'exactitude lors du processus de la conception. Récemment, C et C++ ont été proposés comme base pour créer des spécifications exécutables. Toutefois, ces langages sont conçus pour l'écriture des programmes des ordinateurs, pas pour décrire les ordinateurs ou d'autres composants matériels. Par conséquent, ils ne possèdent pas des fonctionnalités et caractéristiques nécessaires pour décrire les horloges, les signaux, la réactivité et le traitement en parallèle. SystemC explore la première option, il a été récemment mis en place à partir de la librairie des classes C++ pour la conception de la spécification exécutable et de la simulation cycle accurate du hardware en C ++. C'est un support pour les données orientées hardware comme les modules, les ports et les signaux. En réalité, il y avait deux objectifs majeurs dans la conception SystemC :

- ❒ Fournir un seul langage qui permet la vérification des différents systèmes à différents niveaux d'abstraction.
- ❒ Permettre aux concepteurs des systèmes de décrire leurs modèles au niveau RTL, sans les traduire en un langage HDL.

Aujourd'hui, il existe des outils de haut niveau de synthèse des modules SystemC. Ceci a poussé l'industrie à adopter à grande échelle ce langage de conception matérielle-logicielle. En raison de ces caractéristiques, SystemC offre les avantages suivants :

- ❒ La spécification exécutable : un modèle écrit en SystemC peut être compilé et exécuté à la fois.
- ❒ Accélération de la simulation : SystemC est basé sur le langage C++, dont la vitesse de la simulation est élevée par rapport à d'autres langages comme VHDL ou Verilog.
- ❒ Un haut niveau d'abstraction : par rapport à des langages de description matérielle, C++ a la capacité de modéliser des concepts très abstraits de façon élégante. Cette caractéristique est donc intègrée dans SystemC.
- ❒ Implémentation indépendente de l'architecture cible : un modèle présenté dans un langage de description matérielle est généralement spécifique pour une architecture bien définie. Cependant, un modèle décrit en SystemC peut être implémenté soit dans une partie matérielle soit logicielle.

En outre, le simulateur SystemC introduit la notion importante du delta-cycle comme étant l'unité fondamentale de la simulation. L'ordonnanceur (Scheduler) SystemC peut être vu comme un moteur d'évènements : les communications à travers les ports et les canaux, les horloges et les actions des modules sont déclenchés par différents évènements. Le scheduler qui détermine l'ordre d'exécution des processus au sein de l'architecture et ce selon la liste des évènements de sensibilité des processus et les notifications d'événements qui se produisent. La sémantique de ce scheduler a été définie en utilisant les règles des ASM et des sémantiques dénotationnelles. L'unité de base de la simulation est le delta-cycle et une procédure de simulation est donc une séquence de delta-cycles. L'ordonnanceur gère plusieurs tableaux, parmi lesquels nous sommes particulièrement intéressés à la table des processus exécutables (runnable processes : processus qui sont prêts à être exécutés au cours du delta-cycle). Voici une

brève description d'un delta-cycle : un delta-cycle commence lorsque la table des processus exécutables est non vide. L'ordonnanceur exécute ces processus un par un, dans un ordre prédéfini. Chaque processus soit il s'exécute jusqu'à sa fin soit il est suspendu à nouveau (par une commande wait par exemple). Dans le cas où un évènement immédiat est notifié au cours de l'exécution d'un processus, l'ordonnanceur ajoute les processus qui sont actuellement sensibles à cet événement dans la table des processus exécutables. Les delta évènements et les évènements temporisés qui sont générés pendant l'exécution d'un processus seront stockés dans d'autres tables. La table des processus est vidée lorsque tous les processus sont exécutés, et la phase d'exécution de ces processus est appelée une phase d'évaluation. L'ordonnanceur détecte les delta évènements notifiés pendant la phase d'évaluation : s'il ya des processus qui sont sensibles à ces événements, alors il les ajoute à la table des processus. Cette procédure est appelée phase de delta-notification. Si la table des processus est non-vide, l'ordonnanceur entre au prochain delta-cycle et recommence la phase d'évaluation de nouveau. Autrement, il cherche les évènements temporisés qui sont notifiés pendant la phase d'évaluation et ajoute les processus qui sont sensibles à ces événements dans la table des procéssus. C'est ce qu'on appelle la phase de timed-notification. L'ordonnanceur incrémente ensuite le temps de la simulation et entre au prochain delta-cycle. Le processus de la simulation est affecté par l'initialisation des processus, leur exécution et leur ordre, l'activation des événements et les erreurs rencontrées lors de la simulation. La figure ci-dessous montre un diagramme de processus de la simulation en SystemC. En effet, une procédure de simulation peut être considérée comme une succession de delta cycles. Toutes les interactions au sein d'un delta cycle sont abstraites de la perspective de la modélisation. Une telle abstraction est censée fournir une garantie que l'ensemble de ces interactions devrait fonctionner correctement. Autrement dit, les analyses et les vérifications de plus haut niveau peuvent se faire sans prendre en considération ce qui se passe entre les delta cycles. Toutefois, cela est probablement le grand inconvénient de SystemC, comme l'espace du processus final à l'intérieur d'un delta cycle peut être très grand. Un problème typique est le lien de causalité entre les cycles d'attente des processus, qui provoque l'arrêt inattendu du système. En outre, l'accès aux ressources partagées peut mettre en cause des liens de compétitivité entre les processus et, par conséquent, tomber dans un comportement non-déterministe au niveau de delta-cycles. Néanmoins, ceci n'est certainement pas souhaitable dans la conception du matériel.

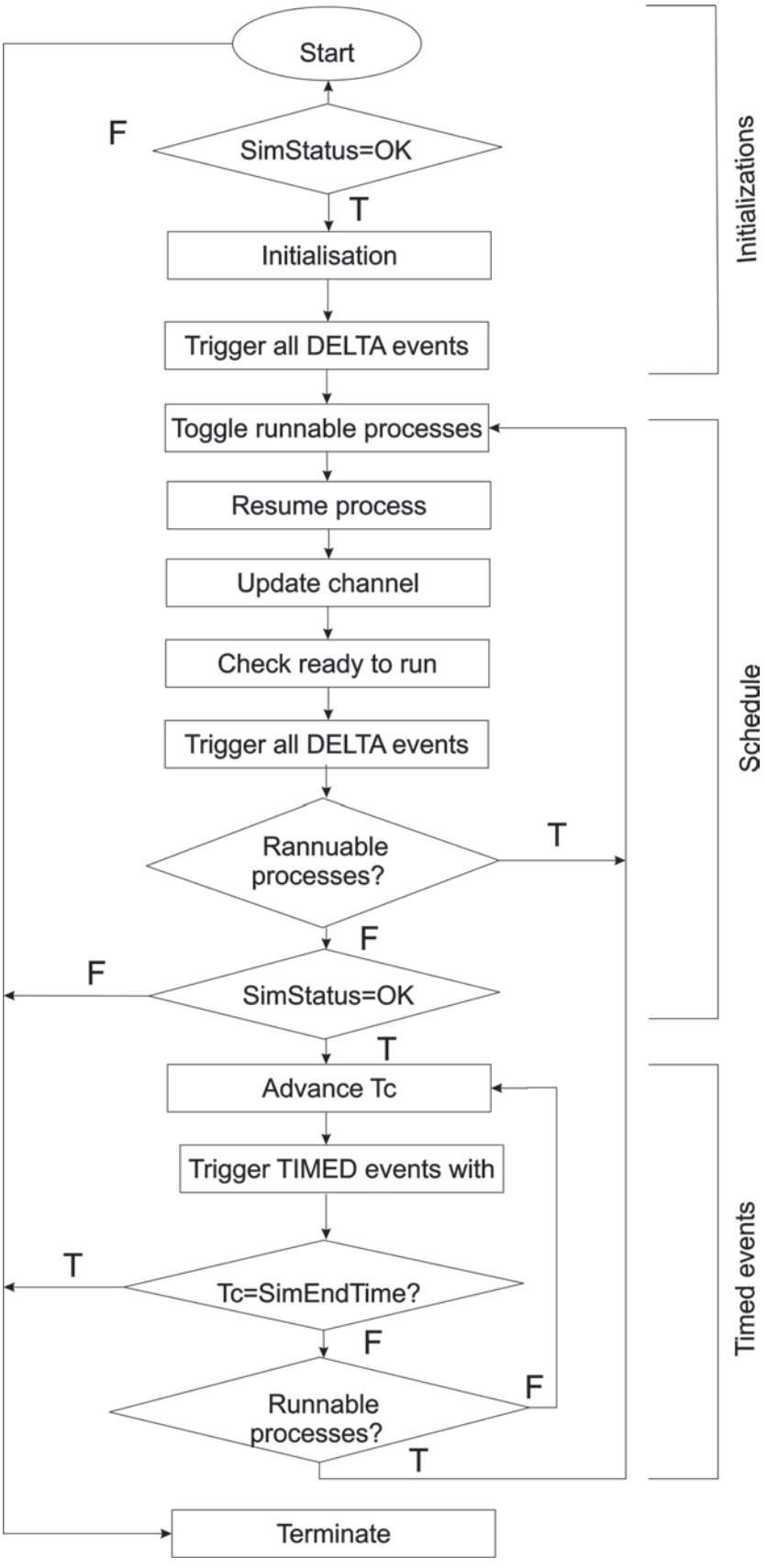


FIGURE 1 – *Algorithme de la simulation de l'ordonnanceur SystemC.*

Pour résoudre ce problème, nous proposons une méthode de modélisation et de vérification des systèmes en utilisant le modèle des SystemC Waiting-State Automata. Ceci est basé sur le fait que la plupart des propriétés importantes dépend fortement de la manière dont les processus passent d'un état d'attente à un autre, qui est en effet contrôlé par l'odonnanceur. Nous proposons d'abord une manière de construire l'automate dit minimal de chaque processus, par l'analyse de ses états d'attente. Ensuite nous définissons des algorithmes pour la composition de l'automate du système global et des algorithmes de la réduction de manière à définir l'abstraction au niveau des delta-cycles. Ces algorithmes sont conformes au simulateur SystemC, qui définit la sémantique de l'exécution du langage. Les vérifications peuvent se faire à la fois au niveau de l'automate de chaque processus et au niveau de l'automate composé du système. Nous discutons aussi quelques extensions basées sur ce modèle, comme l'ajout de compteurs et de temps au niveau des transitions, de sorte que d'autres propriétés peuvent aussi être vérifiées.
Notre approche de vérification a deux objectifs :

- ❒ Assurer que l'introduction d'un nouveau composant ne compromet pas la correction du système initial (sans revérifier le système en entier).

- ❒ Détecter les problèmes liés à la composition des composants du système dont l'exécution peut s'avérer conflictuelle.

Plusieurs tentatives ont été faites pour modéliser des composants SystemC d'une manière formelle (un aperçu sur les travaux connexes est présenté au chapitre 6). Mais chacun d'eux a des restrictions et des limites : soit le modèle proposé décrit SystemC à un niveau d'abstraction plus bas (RTL ou cycle accurate) (par exemple le travail de Drechsler et Grosse [De02] and [De03]). Donc leur modèle ne permet pas de traiter le niveau transactionnel (TLM). Soit, ils ne supportent pas la notion de delta cycle : l'unité fondamentale de SystemC. En effet ces modèles ne peuvent pas faire face à des proprietés telles que la concurrence entre des composants parallèles et le temps continu (par exemple le travail de Kroening et Sharigina [KS05]).

Le modèle du SystemC waiting state automata (WSA), comme présenté initialement par Zhang, Vèdrine et Monsuez dans [YZM07], supprime les contraintes précédemment mentionnées. Nous proposons d'utiliser ce modèle dans une approche bottom-up pour décrire les

composants SystemC au niveau transactionnel et au niveau des delta cycles. Le modèle du SystemC WSA est basé sur l'analyse des instructions wait/notify en SystemC : mécanisme de base qui joue un rôle important dans la simulation en SystemC. Nous avons adopté la modélisation des systèmes complexes en utilisant les automates parce qu'ils permettent de modéliser le parallélisme entre différents composants. Ceci est essentiel surtout pour la description du matériel. Ce choix sera différent si nous modélisons des systèmes distribués composés de quelques composants hétérogènes communiquant en parallèle ou dans le cas de composants séquentiels. Bien que, les réseaux de petris par exemple, sont considérés comme étant plus appropriés pour gérer le parallélisme. Ils ont encore un problème considérable qui est l'explosion combinatoire du nombre d' états du système qui est significativement réduit dans notre modèle. Par ailleurs, d'autres inconvénients des réseaux de petris sont : d'une part, ils ne permettent pas la représentation du système à différents niveaux d'abstraction, plus précisément le niveau des delta-cycles. D'autre part, les réseaux de petris ne permettent pas d'exprimer des propriétés telles que le temps et les compteurs, qui permettent de représenter l'évolution dynamique du système comme on le fait dans le modèle des SystemC WSA.

Dans cette thèse, nous adoptons une approche bottom-up interne basée sur le SystemC WSA en opposition à l'approche top-down (Chapitre 3) : l'approche commence à partir d'une description bas niveau des composants SystemC. Puis elle rassemble tous les composants afin de construire un modèle global pour l'ensemble du système. Mais avant de composer tous les composants ensemble, il est impératif de s'assurer que chaque composant vérifie bien les contraintes spécifiques et qu'il est en mesure d'introduire progressivement les concepts de qualité du service (Qos).

Il y a plusieurs motivations derrière l'utilisation des automates des SystemC WSA pour représenter les composants SystemC : premièrement, il est essentiel de donner une représentation interne de chaque composant du système en utilisant un système de transition d'état. Il est en effet plus facile de vérifier les propriétés sur les composants individuels plutôt que sur l'ensemble du système. Deuxièmement, donner une représentation finie d'un système infini est l'un des récents axes de recherche pour la modélisation des systèmes complexes. En outre, comme mentionné dans [YZM07, HM09, HM12], le modèle de SystemC WSA est conforme à la sémantique de simulation en SystemC car il représente le comportement des composants du système au niveau du delta cycle. En plus le modèle permet de représenter

le système à différents niveaux d'abstraction du niveau système jusqu'au niveau des delta-cycles. Le modèle permet aussi de séparer le comportement interne du comportement global de chaque composant qui est essentiel lors de la modélisation des systèmes parallèles. Ainsi, dans le modèle des SystemC WSA, on considère que les états où les composants sont en communication avec l'environnement. En conséquence, les états internes qui représentent les comportements locaux de chaque composant sont exclus lors de la représentation du système. Contrairement à d'autres modèles formels utilisés pour vérifier les composants SystemC tels que dans [AHT06, KS05, MFM06, KMS06], le modèle des SystemC WSA est différent car il considère que les interactions et les communications entre les processus et la façon dont ils sont gérés par le simulateur SystemC. Il suppose que le comportement d'un processus entre deux états d'attente est abstrait dans le modèle final.Le modèle représente deux informations principales :

- ❒ L'ensemble des conditions d'entrée qui activent et suspendent l'exécution d'un processus et l'ensemble des conditions de sortie qui sont générés.
- ❒ Les points de synchronisation qui représentent les instructions `wait` en SystemC. Ils sont utilisés pour synchroniser entre les processus communicants au niveau des delta-cycles.

L'idée principale derrière le SystemC WSA est de construire un automate pour chaque processus. l'automate est construit à partir de l'ensemble des états d'attente. Il est donc considéré comme une abstraction ou une représentation minimale du programme initial. C'est pour celà, on appelle chaque automate un automate minimal, nous allons utiliser cette notation tout au long de cette thèse.

Le modèle des SystemC waiting-state automata (WSA) est un système de transitions A défini sur un ensemble de variables globales $\mathcal{V}$. C'est un triplet $A = (S; E; \mathcal{T})$, avec S est l'ensemble des états, E est l'ensemble des évènements et $\mathcal{T}$ est l'ensemble des transitions. Chaque tranistion est un 6-uplet $(s; e_{in}; p; e_{out}; f; s')$:

- ❑ s et s' sont deux états dans S, ils représentent respectivement l'état initial et l'état final ;
- ❑ e_{in} et e_{out} sont deux évènements tels que : $e_{in} \subseteq E; e_{out} \subseteq E$;
- ❑ p est un prédicats défini sur les variables dans $\mathcal{V}$, i.e., $FV(p) \subseteq \mathcal{V}$, avec *FV(p)* représente l'ensemble des variables libres dans p ;

❑ f est la fonction définie sur $\mathcal{V}$;

On note $s \xrightarrow[e_{out},f]{e_{in},p} s'$ pour chaque transition $(s; e_{in}; p; e_{out}; f; s')$. L'ensemble des fonctions $\mathcal{F}(A)$ de l'automate $A(V)$ est défini à partir des fonctions dans $A(\mathcal{V})$: $\mathcal{F}(A) = \{f | \exists t \in \mathcal{T} s.t.\ proj_6^5(t) = f\}$, avec $proj_6^5$ est la projection cinquième de la transition dans l'automate. On note aussi $proj_6^1$, $proj_6^2$, $proj_6^3$, $proj_6^4$, $proj_6^6$ qui représentent respectivement l'état initial s, l'évènement d'entrée e_{in}, le prédicat p, l'évènement de sortie e_{out} et l' 'etat de sortie s'.

Ensuite, nous proposons d'étendre le modèle avec des paramètres tels que les informations temporelles et les compteurs. Les automates paramétrés sont utilisés pour étudier divers problèmes de synthèse. Ils sont également utilisés pour modéliser des programmes, dont le comportement dépendra des valeurs des entrées de l'environnement [RAV93]. Les paramètres sont également utilisés pour modéliser les ressources du modèle (tels que le temps, la mémoire) qui sont consommées par les transitions. Nous utilisons tout d'abord la première extension du modèle en utilisant les compteurs comme présenté dans [YZM07]. Les automates avec compteurs sont essentiellement utilisés pour modéliser les systèmes distribués et concurrents et pour vérifier les propriétés comme le *problème d'accessibilité*, *vivacité* et *le déterminisme*. Les auteurs dans [YZM07] utilisent les compteurs pour vérifier d'autres propriétés : déduire les relations entre les conditions d'entrée et les conditions de sortie au niveau de chaque transition du modèle. Dans cette thèse, nous reprenons la même définition des automates avec compteurs comme dans [YZM07]. Mais, nous développons d'avantages l'utilisation des compteurs sur l'automate, et nous spécifions quelques exemples par rapport l'utilisation du paramètre. De plus, nous étendons le modèle avec d'autres paramètres : les informations temporelles, ce qui n'a pas été fait dans le travail précédent. On note respectivement (δ), *(t)* et *(d)* le *compteur*, le temps de début de la transition associée et sa durée. Chaque paramètre est défini sur une transition : (δ) représente le nombre de fois que la transition a été franchie, (t) est le temps de début d'exécution de la transition et (d) est la durée de la transition, une fois déclenchée.

L'idée de notre approche est de définir d'abord un automate minimal pour chaque processus, puis de composer l'ensemble des automates afin de construire un automate pour le système global qui peut être finalement utilisé pour faire du model-checking. Nous procédons tout d'abord à la composition symbolique qui consiste à composer l'ensemble des états d'attente de tous les composants qui sont exécutés en parallèle afin de constuire l'automate du

système global dans une approche bottom-up. En d'autres termes, les composants doivent synchroniser au niveau des états globaux et procéder de façon indépendante au niveau des états locaux. Néanmoins, la composition parallèle des composants SystemC peut provoquer des cycles de causalité entre les procéssus s'exécutant en parallèle . La vérification lors de la composition nécessite en premier lieu de détecter les états dits *unsafe* qui représentent les processus qui sont en attente mutuelle, dans le but d'avoir une analyse plus approfondie basée sur les automates. En outre, la composition symbolique des automates minimaux est récursive au sein de chaque module SystemC.

La composition symbolique des automates est aussi utilisée pour détecter le *déterminisme* au niveau du module : d'abord, on construit l'automate minimal pour chaque composant. Ensuite, tous les automates sont composés ensemble. Si l'automate composé ne contient pas des transitions non-déterministes, on peut confirmer que le modèle est *déterministe*. La détection des transitions non-déterministes peut être effectuée sans faire la composition. On peut simplement vérifier si $f \circ f' = f' \circ f$, où $f \in \mathcal{F}(A); f' \in \mathcal{F}(A')$ (A, A' sont deux automates composés). Cependant, une telle détection n'est pas toujours possible surtout dans le cas où certaines transitions non-deterministes sont jamais déclenchées. En fait, de telles transitions peuvent être supprimées après la composition dans le cadre du raffinement de l'automate composé.

L'algorithme de la compostion symbolique est défini comme suit :

- ❑ $(s_1, s'_1) \xrightarrow[e_{out}, f]{e_{in}, p} (s_2, s'_1) \in \mathcal{T}$" pour chaque état $s_1 \xrightarrow[e_{out}, f]{e_{in}, p} s_2 \in \mathcal{T}$ et $s'_1 \xrightarrow[e'_{out}, f']{e'_{in}, p'} s'_2 \in \mathcal{T}'$, ou bien $e'_{in} \not\subseteq e_{in}$ ou $p \not\Rightarrow p'$.
- ❑ $(s_1, s'_1) \xrightarrow[e'_{out}, f']{e'_{in}, p'} (s_1, s'_2) \in \mathcal{T}$" pour chaque état $s_1 \xrightarrow[e_{out}, f]{e_{in}, p} s_2 \in \mathcal{T}$ et $s'_1 \xrightarrow[e'_{out}, f']{e'_{in}, p'} s'_2 \in \mathcal{T}'$, ou bien $e_{in} \not\subseteq e'_{in}$ ou $p' \not\Rightarrow p$.
- ❑ $(s_1, s'_1) \xrightarrow[e_{out} \cup e'_{out}, f \circ f']{e_{in} \cup e'_{in}, p \wedge p'} (s_2, s'_2) \in \mathcal{T}$" pour chaque état $s_1 \xrightarrow[e_{out}, f]{e_{in}, p} s_2 \in \mathcal{T}$ et $s'_1 \xrightarrow[e'_{out}, f']{e'_{in}, p'} s'_2 \in \mathcal{T}'$.

On a défini aussi les algorithmes de composition pour les automates étendus avec les compteurs et le temps. Ces algorithmes sont basés sur l'algorithme défini précédement mais qui étend aussi l'utilisation des paramètres de l'automate. En effet, on définit des morphismes qui définissent les relations entres les paramètres de l'automate composé en fonction des automates minimaux des composants. On représente ci-dessous les algorithmes de composition

des automates étendus après ajout des différents morphismes :

1. Automates avec compteurs :

   - $\Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta) := \{\delta^*\} \bigcup \Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta)$ et $(s_1, s_1') \xrightarrow[e_{out}, f]{e_{in}, M_c(p), \delta^*} (s_2, s_1') \in \mathcal{T}''$ pour chaque transition $s_1 \xrightarrow[e_{out}, f]{e_{in}, p, \delta} s_2 \in \mathcal{T}$ pour chaque état $s_1' \in S'$ tel que pour chaque transition $s_1' \xrightarrow[e'_{out}, f', s'_2]{e'_{in}, p', \delta'} s'_2 \in \mathcal{T}'$, soit $e'_{in} \nsubseteq e_{in}$ ou $p \not\Rightarrow p'$,
   - $\Pi(s_1', e'_{in}, p', e'_{out}, f', s'_2, \delta') := \{\delta^*\} \bigcup \Pi(s_1', e'_{in}, p', e'_{out}, f', s'_2, \delta')$ et$(s_1, s_1') \xrightarrow[(e'_{out}, f')]{e'_{in}, M_c(p'), \delta^*} (s_1, s'_2) \in \mathcal{T}''$ pour chaque transition $s_1' \xrightarrow[e'_{out}, f']{e'_{in}, p', \delta'} s'_2 \in \mathcal{T}'$ et pour chaque état $s_1 \in S$ tel que pour chaque transition $s_1 \xrightarrow[e_{out}, f]{e_{in}, p, \delta} s_2 \in \mathcal{T}$, soit $e_{in} \nsubseteq e'_{in}$ ou $p' \not\Rightarrow p$,
   - $\Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta) := \{\delta^*\} \bigcup \Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta)$ et $\Pi(s_1', e'_{in}, p', e'_{out}, f', s'_2, \delta') := \{\delta^*\} \bigcup \Pi(s_1', e'_{in}, p', e'_{out}, f', s'_2, \delta')$ et $(s_1, s_1') \xrightarrow[e_{out} \bigcup e'_{out}, f \circ f']{e_{in} \bigcup e'_{in}, M_c(p \wedge p'), \delta^*} (s_2, s'_2) \in \mathcal{T}''$, pour chaque pair de transitions $s_1 \xrightarrow[e_{out}, f]{e_{in}, p, \delta} s_2 \in \mathcal{T}$ et $s_1' \xrightarrow[e'_{out}, f']{e'_{in}, p', \delta'} s'_2 \in \mathcal{T}'$.
   - En ce qui concerne la transition $(s_1, e_{in}, p, e_{out}, f, s_2, \delta_{s_1}^{s_2})$, le morphisme M_c mappe le compteur δ à la *somme* des compteurs des transitions dans $\Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta)$

$$M(\delta) \rightarrow \sum_{\delta^* \in \Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta_{s_1}^{s_2})} \delta^*.$$

2. Automates temporisés :

   - $\Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2) := \{t^*, d^*\} \bigcup \Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)$ et $(s_1, s_1') \xrightarrow[e_{out}, f, d^*]{e_{in}, M_t(p), M_d(p), t^*} (s_2, s_1') \in \mathcal{T}''$ pour chaque transition $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t} s_2 \in \mathcal{T}$ et pour chaque état $s_1' \in S'$ tel que pour chaque transition $s_1' \xrightarrow[e'_{out}, f', d', s'_2]{e'_{in}, p', t'} s'_2 \in \mathcal{T}'$, soit $e'_{in} \nsubseteq e_{in}$ ou $p \not\Rightarrow p'$,
   - $\Pi(s_1', e'_{in}, p', t', e'_{out}, f', d', s'_2) := \{t^*, d^*\} \bigcup \Pi(s_1', e'_{in}, p', t', e'_{out}, f', d', s'_2)$ et $(s_1, s_1') \xrightarrow[e'_{out}, f', d*]{e'_{in}, M_t(p'), M_d(p'), t^*} (s_1, s'_2) \in \mathcal{T}''$ pour chaque transition $s_1' \xrightarrow[e'_{out}, f', d']{e'_{in}, p', t'} s'_2 \in \mathcal{T}'$ et pour chaque état $s_1 \in S$ tel que pour chaque transition $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t} s_2 \in \mathcal{T}$, soit $e_{in} \nsubseteq e'_{in}$ ou $p' \not\Rightarrow p$,

- $\Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2) := \{t^*, d^*\} \bigcup \Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)$ et $\Pi(s'_1, e'_{in}, p', t', e'_{out}, f', d', s'_2) := \{t^*, d^*\} \bigcup \Pi(s'_1, e'_{in}, p', t', e'_{out}, f', d', s'_2)$ et$(s_1, s'_1) \xrightarrow[e_{out} \bigcup e'_{out}, f \circ f', d^*]{e_{in} \bigcup e'_{in}, M_t(p \wedge p'), M_d(p \wedge p'), t^*} (s_2, s'_2) \in \mathcal{T}''$, pour chaque transition $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t} s_2 \in \mathcal{T}$ et $s'_1 \xrightarrow[e'_{out}, f', d']{e'_{in}, p', t'} s'_2 \in \mathcal{T}'$.
- Pour chaque transition $(s_1, e_{in}, p, t^{s_2}_{s_1}, e_{out}, f, d^{s_2}_{s_1}, s_2)$, le morphisme M_t mappe les temps de début t au *min* des temps de début des transitions dans $\Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)$

$$M(t) \rightarrow min_{t^* \in \Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)} t^*$$

- Pour chaque transition $(s_1, e_{in}, p, t^{s_2}_{s_1}, e_{out}, f, d^{s_2}_{s_1}, s_2)$, le morphisme M_d mappe les durées d à la *somme* des durées définies dans $\Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)$

$$M(d) \rightarrow d \geq \sum_{d^* \in \Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)} d^*.$$

3. Automates avec temps et compteurs :

- $\Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2) := \{t^*, d^*, \delta^*\} \bigcup \Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2)$ et $(s_1, s'_1) \xrightarrow[e_{out}, f, d^*]{e_{in}, M_t(p), M_c(p), M_d(p), t^*, \delta^*} (s_2, s'_1) \in \mathcal{T}''$ pour chaque transition $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t, \delta} s_2 \in \mathcal{T}$ et pour chaque état $s'_1 \in S'$ tel que pour chaque transition $s'_1 \xrightarrow[e'_{out}, f', d', s'_2]{e'_{in}, p', t', \delta'} s'_2 \in \mathcal{T}'$, soit $e'_{in} \nsubseteq e_{in}$ ou $p \not\Rightarrow p'$,
- $\Pi(s'_1, e'_{in}, p', t', \delta', e'_{out}, f', d', s'_2) := \{t^*, d^*, \delta^*\} \bigcup \Pi(s'_1, e'_{in}, p', t', \delta', e'_{out}, f', d', s'_2)$ et $(s_1, s'_1) \xrightarrow[e'_{out}, f', d*]{e'_{in}, M_t(p'), M_c(p'), M_d(p'), t^*, \delta^*} (s_1, s'_2) \in \mathcal{T}''$ pour chaque transition $s'_1 \xrightarrow[e'_{out}, f', d']{e'_{in}, p', t', \delta'} s'_2 \in \mathcal{T}'$ et pour chaque état $s_1 \in S$ tel que pour chaque transition $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t, \delta} s_2 \in \mathcal{T}$, soit $e_{in} \nsubseteq e'_{in}$ ou $p' \not\Rightarrow p$,
- $\Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2) := \{t^*, d^*, \delta^*\} \bigcup \Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2)$ et $\Pi(s'_1, e'_{in}, p', t', \delta', e'_{out}, f', d', s'_2) := \{t^*, d^*, \delta^*\} \bigcup \Pi(s'_1, e'_{in}, p', t', \delta', e'_{out}, f', d', s'_2)$ et $(s_1, s'_1) \xrightarrow[e_{out} \bigcup e'_{out}, f \circ f', d^*]{e_{in} \bigcup e'_{in}, M_t(p \wedge p'), M_c(p \wedge p'), M_d(p \wedge p'), t^*, \delta^*} (s_2, s'_2) \in \mathcal{T}''$, pour chaque pair de transitions $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t, \delta} s_2 \in \mathcal{T}$ et $s'_1 \xrightarrow[e'_{out}, f', d']{e'_{in}, p', t', \delta'} s'_2 \in \mathcal{T}'$.

- Pour chaque transition $(s_1, e_{in}, p, t_{s_1}^{s_2}, \delta_{s_1}^{s_2}, e_{out}, f, d_{s_1}^{s_2}, s_2)$, le morphism M_t mappe le temps de début de la transition t au *min* du temps de début défini dans $\Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)$

$$M(t) \rightarrow min_{t^* \in \Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2)} t^*$$

- Le morphisme M_d mappe la durée d à la *somme* des durées définies dans $\Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2)$

$$M(d) \rightarrow d \geq \sum_{d^* \in \Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2)} d^*.$$

- Le morphisme M_c mappe les compteurs δ à la *somme* des durées des transitions définies dans $\Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta)$

$$M(\delta) \rightarrow \sum_{\delta^* \in Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2, \delta_1^{s_2})} \delta^*.$$

Au cours de la composition symbolique, toutes les transitions possibles entre les états symboliques sont générées. Ces transitions contiennent les *transitions sûres*, les *transitions impossibles*, les *transitions redondantes* et les *transitions réductibles*. Définissons tout d'abord chaque catégorie de transitions.

- *Les transitions sûres* : Elles représentent l'ensemble des transitions possibles générés lors de l'exécution symbolique. Ces transitions sont généralement déclenchées à la fois dans les automates minimaux et l'automate composé.

- *Les transitions impossibles* : Elles représentent l'ensemble des transitions qui ne peuvent jamais être déclenchées dans l'automate composé. Elles sont impossibles, soit parce que leurs conditions d'entrée ne peuvent jamais être vraies, ou parce qu'elles correspondent à des états unsafe comme on a expliqué précédemment.

- *Les transitions redondantes* : Elles représentent l'ensemble des transitions qui ont les mêmes conditions d'entrée et les conditions de sortie. Dans ce cas, il est préférable de ne conserver qu'une seule transition.

- ❑ *Les transitions réductibles* : Elles représentent une séquence de transitions consécutives qui sont inter-indépendantes,c'est à dire, les conditions de sortie d'une transition représentent les conditions d'entrée dans la transition consécutive. Dans ce cas, toutes les transitions sont fusionnées ensemble et transformées en une seule transition.

La réduction symbolique est une étape ultérieure qui consiste à garder seulement la trace des transitions sûres. C'est une étape importante pour construire l'automate final. Ainsi, au cours de cette étape on réduit toutes les transitions impossibles, on remplace les transitions redondantes et on gère l'ensemble des transitions réductibles. On considère l'influence de l'environnement sur l'exécution du système, à savoir, éliminer l'ensemble des comportements qui ne peuvent pas se produire dans l'automate composé. En outre, la réduction consiste en la concaténation des transitions, à savoir, l'incidence d'une certaine transition peut immédiatement déclencher une autre transition. On peut donc remplacer les deux transitions par une nouvelle.
L'algorithme de la réduction symbolique est défini comme suit : Supposons un automate $A(V) = (S; E; \mathcal{T})$, telle que $\mathcal{T}$ contient des transitions réductiles, soit $\mathcal{T}_0 := \mathcal{T}$, $\mathcal{T}_{remove} := \{\}$ et $\mathcal{T}_{new} := \{\}$. On définit ci dessous les différentes étapes de la réduction symbolique définies à partir de l'automate composé.

1. pour chaque pair de transitions réductibles (t_1, t_2) et son contractum t_3, avec $t_1, t_2 \in \mathcal{T}_0$, soit $\mathcal{T}_{remove} := \mathcal{T}_{remove} \cup t_1, t_2$ et$\mathcal{T}_{new} := \mathcal{T}_{new} \cup t_3$;

2. on répète les étapes précédentes pour toutes les transitions dans T_0 ;

3. soit $\mathcal{T}_0 := (\mathcal{T}_0/\mathcal{T}_{remove}) \cup \mathcal{T}_{new}$, $\mathcal{T}_{remove} := \{\}$ et $\mathcal{T}_{new} := \{\}$;

4. s'il ya d'autres transitions réductibles dans $\mathcal{T}_0$, on reprend l'étape 1 et on refait le même scénario ; sinon, soit $\mathcal{T}' := \mathcal{T}_0$.

L'automate réduit est donc $(S, E, \mathcal{T}')$.
Notons qu' à ce stade, les évènements de l'automate final peuvent être divisés en deux ensembles : l'ensemble des événements provenant de *l'environnement* E_e et l'ensemble des évènements internes E_i. Les événements de l'environnement sont des événements générés par le simulateur SystemC, qui sont généralement des évènements temporels tels que les

événements liés aux horloges. En utilisant la dernière classification des événements, l'automate composé peut être réduit à nouveau en supprimant ces transitions dont les évènements sont liés à l'environnement, c'est à dire, les transitions où $e_{in} \notin E_e$.

Les automates des SystemC WSA sont des systèmes de transition qui sont extraits manuellement à partir des descriptions SystemC [YZM07]. Donc il faut définir un ensemble d'étapes de construction automatique du modèle ; ceci est la principale contribution de cette thèse. On a aussi prouvé que le modèle est conforme au système initial, puis on l'a validé durant l'étape de la construction automatique. Pour construire les automates abstraits, nous suivons

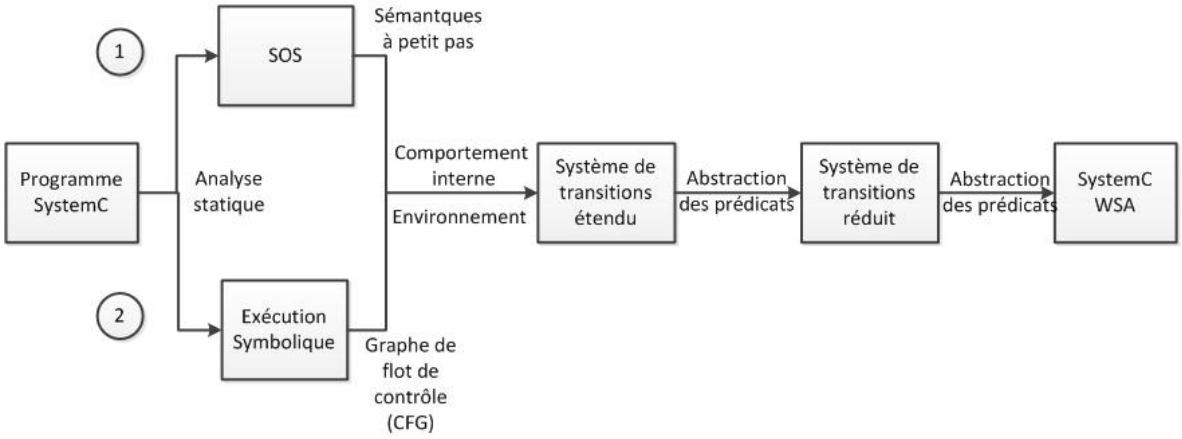


FIGURE 2 – *Les étapes de construction automatique des SystemC WSA*

les différentes étapes comme indiqué dans la Figure 2 : (1) Nous avons besoin d'écrire correctement la sémantique formelle du langage SystemC. Nous avons ainsi utilisé une sémantique à petit pas appelée sémantique opérationnelle de Plotkin [Plo04]. Le but de développer une telle sémantique est (*i*) de fournir une description complète et non ambigue du langage, (*ii*) d'exécuter pas à pas le programme initial et (*iii*) de détecter l'effet de cette analyse sur le comportement gloal du système. On distingue aussi entre le comportement interne et le comportement global de chaque module SystemC. Toutes ces informations sont présentées dans la syntaxe de la sémantique opérationnelle du programme. Nos sémantiques capturent non seulement la structure des composants SystemC, mais aussi le comportement de la composition parallèle des composants communicants. Nous avons donc modélisé aussi le comportement de l'ordonnanceur. Nous supposons que chaque module se comporte soit localement en utilisant ses variables locales ou communique avec l'environnement à travers ce qu'on appelle

les variables d'environnement. Les *variables locales* sont des signaux de sortie, des variables internes, des canaux de sortie, les événements de sortie, et le compteur de programme pour les processus. Les variables *d'environment* sont des signaux d'entrée, les événements d'entrée, les canaux d'entrée et les variables globales. En ce qui concerne la sémantique de la simulation en SystemC[WRM01], il existe au plus un processus qui réagit avec l'environnement. Nous pouvons visualiser localement les instants au cours desquels les réactions se produisent, en observant l'état (les variables C de et les compteurs ordinaux pour chaque processus) du procéssus, noté σ et son environnement (événements, les canaux, les signaux, les processus, etc), noté E. Pour décrire comment une instruction modifie les configurations de l'environnement, nous écrivons nos règles comme suit :

$$\langle stmt, \sigma \rangle \xrightarrow[E_o]{E} \langle stmt', \sigma$$

où :

- stmt est une instrcution SystemC qui correspond à l'emplacement du compteur de programme, avant la transition, et *stmt'* est l'emplacement du compteur du programme après la transition,
- σ et σ' sont respectivement l'état initial et l'état final au cours de la réaction. Ils représentent une fonction $\mathcal{V} \cup \mathcal{CH} \mapsto values$, où $\mathcal{V}$ est l'ensemble des variables locales et partagées et $\mathcal{CH}$ est l'ensemble des canaux.
- E est l'environnement (ensemble des événements et des variables qui activent le processus) dans lequel la transition s'est exécutée. E_o est l'environnement émis en sortie pendant la transition. En général, un environnement est un 5-uplet $E = (E^i, E^\delta, E^T, \mathcal{V}, \mathcal{RQ})$ où :

❑ E^i est l'ensemble des événements immédiats,

❑ E^δ est l'ensemble des prochains delta événements,

❑ E^T est l'ensemble des Timed événements,

❑ $\mathcal{V}$ est l'ensemble des delta mises à jour des variables.

- ❑ $\mathcal{RQ}$ est une séquence constituée des demandes en attente de mise à jour les canaux. Une demande est une paire $(ch, exp(\sigma))$ où $ch \in \mathcal{CH}$ et $exp(\sigma)$ représente la valeur attribuée à ch.

Pour indiquer que l'environnement de sortie E_o ne change pas, on utilise la notation suivante $-$. Soit, il y a pas d'événements émis au cours de la transition, soit les variables restent inchangés ou bien les canaux ne sont pas modifiés. Notre sémantique est similaire à celle de Shyamasundar[RSK07] où une sémantique complète du langage SystemC est proposée. Dans notre approche, nous insistons particulièrement sur trois points principaux :

- capturer le comportement réactif lors de la simulation en SystemC.
- spécifier un réseau de composants synchrones et asynchrones qui communiquent soit à un très haut niveau ou bien via les événements à un niveau plus bas.
- spécifier deux niveaux de temps : le cycle de delta et le temps de simulation.

En outre, au cours de notre formalisation et surtout pendant la composition parallèle, nous distinguons entre les trois phases du processus de simulation (en réponse à l'algorithme de simulation en SystemC (Figure 1) : C'est une principale contribution de notre sémantique. Nous intègrons la simulation du scheduler dans une composition parallèle de processus concurrents. Cette composition est indépendante de l'ordonnanceur lui-même. L'ordonnanceur est alors dispensé du processus de modélisation. Il est déjà présenté principalement dans la composition parallèle.

Nos sémantiques ont deux avantages principaux : d'abord, elles commencent à partir d'une description de bas niveau du composant SystemC (au niveau du delta cycle), qui met en évidence l'évolutivité de l'approche globale. Deuxièmement, nous ne devons pas modéliser séparément l'ordonnancer. Ainsi, l'automate composé sera généré indépendant de la politique d'ordonnancement ce qui nous permet de gagner en termes de coût de modélisation et de vérification. Nous allons présenter la sémantique de certaines constructions séquentielles (y compris les affectations, les instructions liées des canaux, les instructions liées aux événements, les instructions conditionnelles, les instructions wait). Nous présentons aussi les sémantiques de la composition parallèle où l'on distingue entre les trois étapes de la sémantique SystemC qui est la principale contribution de notre travail par rapport aux travaux existants en formalisation de la sémantique du SystemC utilisant les notations SOS.

assignment	$< var\ v, \sigma > \xrightarrow[-]{-} < \epsilon, \sigma[v] >$
if	$\frac{<b,\sigma> \rightarrow <true,\sigma> \quad <p,\sigma> \rightarrow <\epsilon,\sigma'>}{<if\ b\ then\ p\ else\ q,\sigma> \xrightarrow[-]{-} <\epsilon,\sigma'>}$ $\frac{<b,\sigma> \rightarrow <false,\sigma> \quad <q,\sigma> \rightarrow <\epsilon,\sigma'>}{<if\ b\ then\ p\ else\ q,\sigma> \xrightarrow[-]{-} <\epsilon,\sigma'>}$
while	$\frac{<b,\sigma> \rightarrow <true,\sigma> \quad <p;\ while\ (b)\ do\ p,\sigma> \rightarrow <\epsilon,\sigma'>}{<while\ (b)\ do\ p,\sigma> \xrightarrow[-]{-} <\epsilon,\sigma'>}$ $\frac{<b,\sigma> \rightarrow <false,\sigma>}{<while\ (b)\ do\ p,\sigma> \xrightarrow[-]{-} <\epsilon,\sigma>}$
Les canaux	$\frac{ch \in Channels \wedge \sigma(ch) \neq exp(\sigma)}{\langle ch!!exp,\sigma\rangle \xrightarrow[E^I,E^\delta,E^T,\mathcal{V},\mathcal{RQ}\setminus(ch,exp(\sigma))]{E^I,E^\delta,E^T,\mathcal{V},\mathcal{RQ}} \langle \epsilon,\sigma[v/exp]\rangle}$ $\frac{ch \in Channels, v \in \mathcal{V}}{\langle ch??exp,\sigma\rangle \xrightarrow[-]{E} \langle \epsilon,\sigma[v/ch]\rangle}$
notify	$\langle e.notify(),\sigma\rangle \xrightarrow[e,e,\emptyset,\emptyset,\emptyset]{E} \langle \epsilon,\sigma\rangle$ $\langle e.notify\_\delta(),\sigma\rangle \xrightarrow[\emptyset,e,e,\emptyset,\emptyset]{E} \langle \epsilon,\sigma\rangle$ $\langle e.notify\_t(),\sigma\rangle \xrightarrow[\emptyset,\emptyset,e,\emptyset,\emptyset]{E} \langle \epsilon,\sigma\rangle$
wait	$\frac{e \notin E}{\langle wait(e),\sigma\rangle \xrightarrow[-]{E} \langle wait(e),\sigma\rangle}$ $\frac{e \in E}{\langle wait(e),\sigma\rangle \xrightarrow[-]{E} \langle \epsilon,\sigma\rangle}$

TABLE 1 – Sémantiques opérationnelles de quelques instructions SystemC

(2) Nous procédons à des techniques d'exécution symbolique (SE) [Kin76] pour générer le graphe de flot de contrôle du programme. Nous l'appelons `exécution symbolique conjointe` car nous combinons à la fois l'exécution symbolique et la sémantique opérationnelle. Les principaux objectifs de l'application de l'exécution symbolique sont : générer premièrement les différentes traces d'exécution du système et deuxièmement exprimer le programme en utilisant des formules logiques à la place des expressions réelles. Cette étape est une étape primordiale pour appliquer les techniques d'abstraction des prédicats qui représentent l'étape suivante. L'exécution symbolique est une extension de l'exécution réelle ayant pour paramètres d'entrée l'ensemble des opérateurs de base de langage exprimés sous forme de prédicats et comme sortie un ensemble de formules symboliques définis sur ces prédicats.
Prenons comme exemple un programme écrit dans un langage de programmation quelconque, on suppose que les variables du programme sont de type entiers signés et que le programme contient des instructions simples type IF (avec des clauses THEN et ELSE), des instructions GO-TO et des entrées (paramètres de procédure, variables globales, operations read). On prend comme expressions arithmétiques les opérateurs basiques pour les entiers comme l'addition (+), la soustraction (−) et la multiplication (x). On suppose qu'une expression booléenne utilisée dans l'instruction IF est un simple test supposant que l'expression arithmétique est non-négative ($i.e. arith.expr \geq 0$). L'exécution symbolique de ce programme consiste à tranformer cette description sous forme de symbôles mathématiques sans toucher ou changer la sémantique du programme. On suppose qu'à chaque fois une nouvelle valeur est demandée, elle sera fournie à partir de la liste des symbôles suivants : $(a_1, a_2, \ldots, a_n)$. Puis selon la nature de l'instruction correspondante à chaque ligne du code, on associe une fontion algébrique définie sur ces variables. Le paramètre état d'exécution d'un programme correspond aux valeurs des variables et le PC (path condition). Il pointe sur l'instruction en cours d'exécution.

(3) Nous procédons à des techniques d'abstraction de prédicats (PA)[FQ02] pour déterminer tout d'abord les relations entre les formules logiques générées pendant l'exécution symbolique des automates parallèles. Ensuite, ces techniques permettent de fusionner les chemins entre chaque deux états d'attente dans le graphe de flot de contrôle. Le but final est de construire l'automate du SystemC WSA à partir du graphe de flot de contrôle qui est annoté avec des formules logiques définies à partir des variables globales et des informations sur les

événements de l'environnement.

Enfin, nous proposons d'utiliser le modèle des SystemC waiting state automata dans trois applications diffèrentes. Tout d'abord, nous présentons une approche globale pour modéliser et simuler symboliquement les systèmes embarqués logiciels et matériels en utilisant les SystemC WSA (comme présenté dans la figure ci-dessous). Nous montrons que notre approche garantit

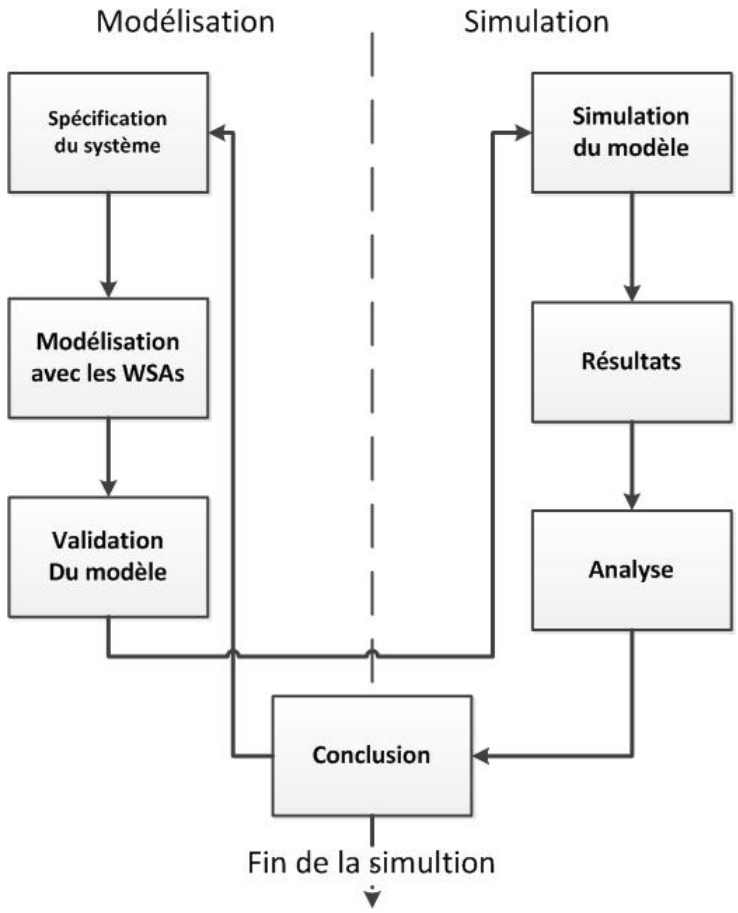


FIGURE 3 – *La modélisation et la simulation symbolique des composants SystemC à l'aide des SystemC WSA*

une simulation rapide des systèmes embarqués. Ensuite, nous présentons notre méthodologie pour calculer et estimer le pire temps d'exécution (WCET) en utilisant le modèle des Timed SystemC WSA et comparer la méthodologie à des méthodologies existantes. Nous utilisons le modèle des Timed SystemC WSA pour modéliser le matériel, puis nous exécutons symboliquement le programme sur le modèle abstrait. Nous procédons à une technique de fusion intelligente présenté dans des travaux éxistants dans notre équipe pour donner une estima-

tion précise du WCET. Cette application est un travail conjoint qui réunit deux domaines de recherche à l'ENSTA ParisTech. Nous avons ensuite proposé d'utiliser des techniques de vérification notamment les techniques du *model checking* pour vérifier d'autres propriétés sur le modèle des SystemC waiting state automata. Nous énumérons aussi les principaux anomalies qui se produisent en raison de la concurrence et de l'accès aux ressources partagées entre les composants s'exécutant en parallèle. Nous proposons une solution pour éviter ces anomalies en utilisant les automates des SystemC WSA.

(La phase d'évaluation)

(1)

$$\frac{\forall i{\in}\{1..n\},\exists e{\in}E^I,waiting(P_i,e)\ \wedge\forall j{\in}\{n{+}1..m\},\forall e{\in}E^I,\neg waiting(P_j,e)}{\langle P_1\|\cdots\|P_n\|\cdots\|P_m,\sigma\rangle \xrightarrow[(\emptyset,E^\delta,E^T,V^\delta,\mathcal{RQ})]{E} \left\langle P_1'\|...P_n'\|\cdots\|P_m,\sigma'\right\rangle}$$

(2)

$$\forall i \in \{1\dots n\},\ waiting(P_i)\ \ \forall j \in \{n+1\dots m\}, ready(P_j)$$

$$select\ p \in \{n+1\dots m\}, \langle P_p,\sigma\rangle \xrightarrow[(E_p^I,E_p^\delta,E_p^T,V_p^\delta,\mathcal{RQ}_p)]{E} \left\langle P_p',\sigma'\right\rangle$$

$$\frac{add(\left\langle E_p^\delta,E^\delta\right\rangle,\left\langle E_p^T,E^T\right\rangle,\left\langle V_p^\delta,V^\delta\right\rangle)}{\langle P_1\|\cdots\|P_n\|\cdots\|P_p\|\cdots\|P_m,\sigma\rangle \xrightarrow[(E_p^I,E_p^\delta\cup E^\delta,E_p^T\cup E^T,V_p^\delta\cup V^\delta,\mathcal{RQ}_p\cup\mathcal{RQ})]{(E^I,E^\delta,E^T,V^\delta,\mathcal{RQ})} \left\langle P_1'\|\cdots\|P_n'\|\cdots\|P_p'\|\cdots\|P_m,\sigma'\right\rangle}$$

(La phase de mise à jour)

$$\frac{\forall (ch,v){\in}RQ}{\langle P_1\|\cdots\|P_n,\sigma\rangle \xrightarrow[(E^I,E^\delta,E^T,V^\delta,\emptyset)]{(E^I,E^\delta,E^T,V^\delta,\mathcal{RQ})} \langle P_1\|\cdots\|P_n,\sigma[v/ch]\rangle}$$

(La phase des delta-cycles)

$$\frac{\forall i{\in}\{1..n\},waiting(P_i)}{\langle P_1\|\cdots\|P_n,\sigma\rangle \xrightarrow[(E^\delta,\emptyset,E^T,\emptyset,\mathcal{RQ})]{(\emptyset,E^\delta,E^T,V^\delta,\mathcal{RQ})} \left\langle P_1\|\cdots\|P_n,\sigma[V^\delta/V]\right\rangle}$$

(La phase d'avancement du temps)

$$\frac{\forall i{\in}\{1..n\},waiting(P_i)}{\langle P_1\|\cdots\|P_n,\sigma\rangle \xrightarrow[(E^T,\emptyset,\emptyset,\emptyset,\mathcal{RQ})]{(\emptyset,\emptyset,E^T,\mathcal{RQ})} \left\langle P_1\|\cdots\|P_n,\sigma[V^\delta/V]\right\rangle}$$

TABLE 2 – Semantiques Opérationnelles relatives à la Composition Parallèle

Contents

CHAPTER 1

Introduction

Nowadays, embedded systems are more and more integrated in critical applications such as : automobile, avionics, satellites, telecommunications, medical equipments, etc. They are usually composed of deeply integrated but heterogeneous hardware and software components. Those components are developed under severe **resource limitations** (i.e, small processors, tiny memory and low power) and under **high quality requirements** (i.e, speed, real-time constraints, accuracy, consumption and an operational long life-cycle). As a consequence, the job of design engineers has become more tricky and challenging, due to the intensive increasing gap between the cost, the embedded functions, and the performance of those systems.

To meet the high quality standards in nowadays embedded systems and to satisfy the rising industrial demands, the automatization of the developing process of those systems is gaining more and more importance. A major challenge is to develop an automated approach

that can be used for the integrated verification and validation of complex and heterogeneous HW/SW systems.

1.1 Context

Traditionally, embedded systems were developed by separating the hardware part from the software part. It takes several iterations in the design process to reach an implementation that is functionally correct and that satisfies the performance requirements. Those iterations consume large amounts of costly development time, especially because they occur in a phase where the design is already implemented with a lot of details involved. Yet, this technique is no long appropriate for nowadays embedded system design due to market pressure that require **quick**, **valid**, **efficient** and **safe** systems.

Thus, due to the design trends mentioned above, new modeling languages that support both hardware and software co-design have emerged. Among others, we mention the SystemC language[sys], which is a system level design language that supports design space exploration and performance evaluation efficiently throughout the whole design process even for large and complex HW/SW systems. SystemC is a C++ based modeling framework that supports system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis.

SystemC allows the description of both hardware and software parts. Besides, it allows to execute designs at different levels of abstraction. As a consequence, co-simulation, which can be defined as the simultaneous execution of hardware and software, is used to **verify** and **validate** the embedded system throughout the whole design process. However, co-simulation is necessary because it ensure that models are stepwise refined throughout the conceptual design, but still be *not sufficient* : First, because it cannot cover all possible execution scenarios in particular for real-time, non deterministic and non-terminating systems. Second, it is very difficult to ensure the consistency between different abstraction levels, or to reuse verification results in later development stages. Finally, and more precisely, the evaluation of the simulation results should be done manually by the designer, which needs to be computed automatically. Due to the previous limitations, we need to exploit new methodologies for program analysis and verification, to help designers detect and correct errors in early stages of the conceptual design before proceeding to implementation.

Several attempts have been made to model SystemC designs in a formal way (an overview about related works is presented in Chapter 6). But each of them has some restrictions and limitations : either the model they propose describe SystemC designs at a low level (RTL or cycle accurate) (e.g work of Drechsler and Grosse [De02] and [De03]) and does not treat the transactional level (TLM) ; or, they don't support the notion of delta-cycle : the fundamental unit of SystemC scheduler and so they can't deal with properties like concurrency and continuous time (e.g work of Kroening and Sharigina [KS05]).

The SystemC waiting-state automaton (WSA), as first presented by Zhang, Védrine and Monsuez in [YZM07], removes these constraints. We propose to use this model in a compositional bottom-up approach for describing SystemC designs at the **transaction level** within a **delta-cycle**. In this thesis, we also propose to develop and extend the model as presented in different works [HM09, HM12]. The proposed model succinctly captures the reactive features of SystemC components communicating through either high level transactions or low-level signal and event communications. It also elucidates the anomalies that are introduced by the simulation kernel of SystemC due to concurrency between components and stresses on the observable behavior of processes that influence the simulation procedure due to the SystemC *wait/notify* mechanism. The proposed approach is basically a bottom-up approach which requires refinement during composition. We define first several steps to build and apply the approach and we illustrate that on several examples. We then propose different applications of the approach towards the verification of functional and non-functional properties of SystemC designs.

1.2 Summary of Contributions

During the thesis, we present the past and the latest works on the formal verification and modeling of embedded systems especially those written in system languages like **SystemC**. We also present the limitations of those works compared to our approach. Those works address the same class of problems that we do in this thesis. But we propose here a new, complete and efficient solution to study embedded systems and critical problems related to the execution time and concurrent access to shared resources. In particular, we propose a new abstract model that is used to represent any architecture independently from its internal implementation and that is extended with further parameters in order to verify further properties.

1.2.1 Research Questions

This thesis has a single main research challenge. To achieve this main challenge,we break it down in three sub-challenges that we will highlight throughout this thesis. The context for those challenges is to propose a stepwise approach to automatically build an abstract representation (model) of hardware/software systems that is used to verify not only the functional properties but also non-functional properties of those systems. We also propose to use this abstract representation in a global approach in order to model and analyze the behavior of embedded complex systems written in a system level modeling language.

☞ **Main-challenge** : How to accurately describe both the functional and the temporal behavior of complex embedded systems ?

✍ **Sub-challenge 1** : What kind of abstract model should be used to represent the behavior of the embedded system ? and at which level of abstraction regarding to the properties we want to verify ? These two constraints are closely coupled and form a basic trade-off.

✍ **Sub-challenge 2** : What methods are suitable to extract and build the abstract model from a complex embedded system containing both the hardware and the software ?

✍ **Sub-challenge 3** : What methods are suitable for validating models describing the temporal and functional behavior of complex embedded systems ?

1.2.2 Contributions

The main contributions of this thesis compared to existing works are as follows :

❑ **Modeling the behavior of the embedded system using automata.** The thesis represents a bottom-up approach based on the SystemC waiting-state automaton : an abstract automaton used to model both the hardware and the software. The particularity of this model is that it substantially reduces the state space of the system under study. It is also consistent with the simulation semantics SystemC language. Besides, it is capable to handle both functional and non-functional properties of the system under study.

❑ **Validating the model.** In order for a model to be useful, it must be assured that the model is valid, i.e. we prove that it is an accurate description of the intended system at

the appropriate level of abstraction. Validation is ensured stepwise during the process of building the abstract automaton of each component. In fact, we define first a complete and an accurate definition of the semantics of a subset of the system-level language under study, we then proceed to techniques like symbolic execution and predicate abstraction to automatically build the abstract model.

- **Determining non-functional properties.** In a Real-Time Systems, designers must ensure that the results of the computations are logically correct with respect to the physical instant at which those results are produced : this is what we call hard real time constraints. Thus, a missed deadline in hard real-time systems is catastrophic and in soft real-time systems it can lead to a significant loss. Hence, an exact estimation of the system timing behavior is the most important concern in these systems. Timing analysis is in general performed from two levels : *Worst-case execution time (WCET)* analysis and the *Higher-level/system-level analysis*. In both levels we must ensure that our formal analysis gives an exact approximation of the system execution time without loss of precision.

1.3 Outline

This thesis is structured as follows : In Chapter 2, we study the concept of System-On-Chip design process including different steps of the conceptual design, SoC bottlenecks and we briefly compare the traditional and the new methodology for the hardware SoC design. In Chapter 3, we discuss two alternative design approaches adapted for systems modeling and validation. We present the top-down approach and the bottom-up approach, we present their advantages and drawbacks and we compare them to our approach. Chapter 4 is devoted to the abstraction levels in the conceptual design, we present different levels of abstraction and their degree of granularity compared to the initial system. We stress on the need to raise the level of abstraction in order to give an abstract representation of the system without giving more details about its implementation.

In Part I, we first introduce the *transaction level modeling (TLM)*. Next, in Chapter 5, we introduce the SystemC language : its syntax, its structure and basically the simulation semantics of its **scheduler**. We specifically stress on its ability to model high-scaled complex

systems and to handle different levels of abstraction including the TLM.

In Part II, we present the SystemC waiting state automata (WSA) as a new abstract representation for SystemC modeling, that is conform to the simulation semantics of SystemC at both the TLM and the delta-cycle level. This model was first proposed in [YZM07]. The main drawback of this paper, as mentioned in [PHG08], is that the model has to be build manually. In this thesis, we first propose to extend the model with further parameters and to express more in details the usefulness and the idea behind the model. Second, we propose a stepwise approach on how to extract and build automatically the model from SystemC designs. To do so, we give a detailed description of the terminology of the abstract model, we compare it to existing approaches for SystemC modeling and we enumerate our main contributions compared to them. Later, we present the possible extensions of the SystemC WSA, among others we mention the time and counters parameters and we present how to use each parameter in our analysis process. We also present algorithms to symbolically compose and reduce automata in order to build the automaton for the global system. In the end of this part, we propose how to generate the abstract model by separating threads from methods, we define the algorithm for each process and we illustrate that on some examples.

In Part III, we proceed to the automatic generation of the abstract automata from SystemC designs. We start by defining a clear and detailed description of the semantics of a subset of SystemC using the structural operational semantics (SOS) of Plotkin. The particularity of our semantics is that we distinguish between the three phases of the simulation semantics of SystemC scheduler : the **evaluation** phase, the **delta-cycle advancing** phase and the **simulation time advancing** phase. Our semantics capture not only the structure of SystemC components but also the **compositional** and **reactive** behavior of the communicating components. Next, we proceed to the symbolic execution to generate the control flow graph (CFG) of the program, we define a particular symbolic execution that we call *conjoint symbolic execution* that generates not only the CFG but also small step semantics of the program using the SOS. Finally, we resort to predicate abstraction to reduce and abstract the control flow graph and consider only specific states, i.e, only states where a process is visible to (or communicating with) its environment. This final step is essential to build the final SystemC waiting-state automaton.

Part IV is dedicated to real applications of the SystemC waiting-state automata. First, we

present a global framework to model and symbolically simulate software/hardware embedded systems using the SystemC waiting-state automata. We prove that our framework garantee a fast simulation of the embedded system. Next, we present our methodology to compute and estimate the worst-case execution time (WCET) using the Timed SystemC WSA model and compare it to existing methodologies. We use the Timed SystemC WSA to model the hardware and then we symbolically execute the program on it, we proceed to a smart fusion technique presented in previous work in our team to give an accurate estimation of the WCET. This application is a joint work that brings together two areas of research at ENSTA ParisTech. We then, propose a framework on how to use verification techniques notably model checking techniques to verify further properties on the SystemC waiting-state automata. We also enumerate the main anomalous behaviors that occur due to the concurrency and the access to shared resources between SystemC components. We propose a solution how to avoid those anomalies using The SystemC waiting-state automata.

In Part V, we resume the main contributions of this thesis and we give an outlook on further research topics.

CHAPTER 2

The System-on-Chip Design Flow

2.1 Introduction

Systems-on-chips (SoCs) are simply electronic systems that are implemented on a single chip. This technology, despite its little size, reduces the time to market constraints and the developing process of real time applications but also increases the performance of the whole system. With the appearance of the CMOS (complementary metal oxide semiconductor) technologies, hardware and software applications can be embedded on the same chip. Despite this and due to Moore's law [Lei05], that have significantly influenced the evolution of system design, designers of System-on-Chips are facing an increasing productivity gap between the technology used and the tools supported to verify the SOC.

A lot of efforts have been made to raise the level of abstraction in the design process. The aim is to represent the system with less and enough details so that to improve the **accuracy** and the **efficiency** of the simulation speed. We often talk about the `SOC flow design`; it is a methodology to represent a hardware/software system at different levels of abstraction starting from the specification of the application toward the implementation of the software on the platform. The flow design consistency is dependent on the complexity of the SOC itself. This complexity may lead to several bottlenecks that designers are trying to avoid [ed05b].

2.2 SoC Bottlenecks

The system design (Figure 2.1) starts from a set of requirements and constraints used to identify and describe different parts of the system. Requirements are then expressed as specifications. As shown in Figure 2.1, the design is split into two parts : (a) express the specification of the computational part into a set of processors or components and (b) express the specifications of the communication part into the set of buses. During different steps of the design flow, we first allocate components, then we partition specifications between the components and finally we schedule the execution of different components. Later, components are implemented by hardware/software synthesis.

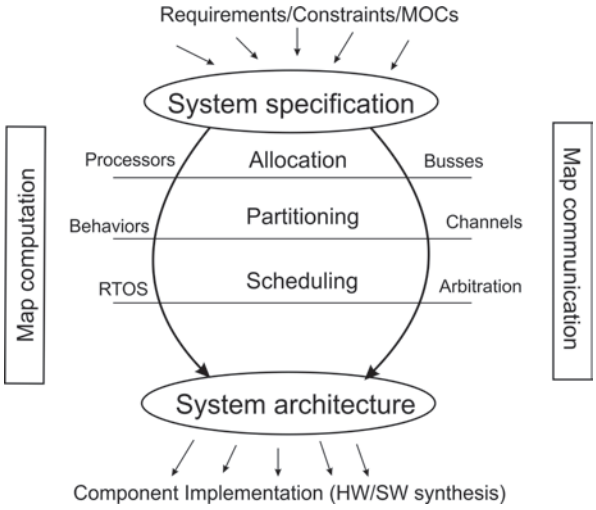


FIGURE 2.1 – *System design tasks [ed05b]*

Challenges in the design of embedded systems must satisfy the following constraints :(i) increasing application complexity ; (ii) increasing target system complexity and (iii) finding

the right balance between different constraints resulting from this complexity : cost, power consumption, timing constraints, dependability.

Figure 2.2 shows the potential gap that exists between improvements made in design productivity and devices integration. The *design productivity gap* is defined by the International Technology Road map for Semiconductors (ITRS) [ITR] as the difference between what is possible to **manufacture** and what is possible to **design**. Thus, while transistors are growing in a logarithmic rent due to Moore's Law [Lei05], productivity is less growing due to the lack of adequate tools that support this growth.

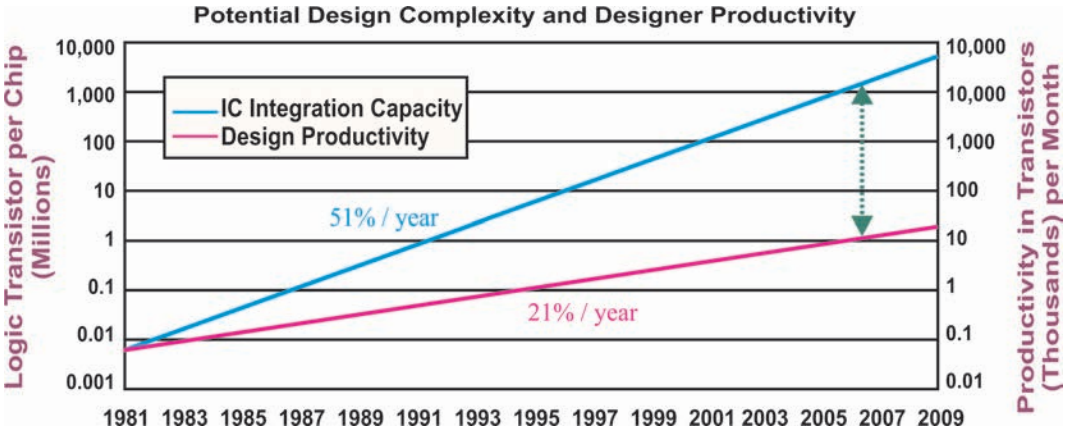


FIGURE 2.2 – *Design Productivity Gap (source : SEMATECH)*

We resume now the three major bottlenecks in SoC design as mentioned in [ed05b].

2.2.1 Explosive Complexity

Complexity is one of the most remarkable bottleneck of nowadays embedded systems since we are integrating more functions that perform more tasks in one and small system. Hence, a typical SoC integrates many blocks including peripheral IPs, multiple processors, memory cuts, buses, complex interconnects, etc. We don't have only to manage the integration of these components in the same chip but also manage the interactions and communications between them. For all of these reasons, designers are trying to reduce system complexity at different levels of system design. But, this is unfortunately a very tough and time-consuming job to cope with.Hence, the traditional design flow is no longer used for nowadays embedded systems since it doesn't deal with the complexity issue.

2.2.2 Time-to-Market Pressure

Time-to-Market is the amount of time required to find a solution for the initial requirements and implement it on a final product that is functionally correct. Nevertheless, the increasing complexity of current SoC products usually necessitates time-consuming development phases. Besides, the classic design flow is unfortunately affecting the time to market since we have to wait too long to have an available prototype.

2.2.3 Cost

The ever-increasing cost of SoC development and production is also one of most intricate problem in the SoC industry. This is why, errors in the design functionality is no more tolerated. Costs cover the design process, the technology used for that and the set of tools used for the verification and the manufacturing of the SoC itself. The traditional design flow is also not able to solve the problem of the cost rise.

2.3 Traditional vs New SoC Design Flow

In this section, we study the evolution of the methodologies used in the conceptual design of complex systems : the traditional design process and the new design process [ed05b].

In a traditional design process, the system specification is directly followed by hardware and software development. This is why, it is not easy to reach an implementation that is functionally correct and that respects the initial specifications. We may need to follow several steps and iterations to produce a final SoC that is conform to the specifications. As a result, the traditional design flow is a time consuming and an inefficient process since it is not usually easy to add any transformations to the design after prototype. The traditional design flow is illustrated in Figure 2.3. As shown in this figure, the system specification is split into two distinct parts : (1) system hardware development and (2) system software development. Note that there is no communication between these two parts. Each part is developed independently until we generate a prototype of the system under design. Thus, no transformations can be added once the hardware and the software are designed. The traditional hardware design process relies classically on three different levels of abstraction. A general classification of the design process is available through the Y-Chart [GK83, AG02](Figure 2.4). It defines system

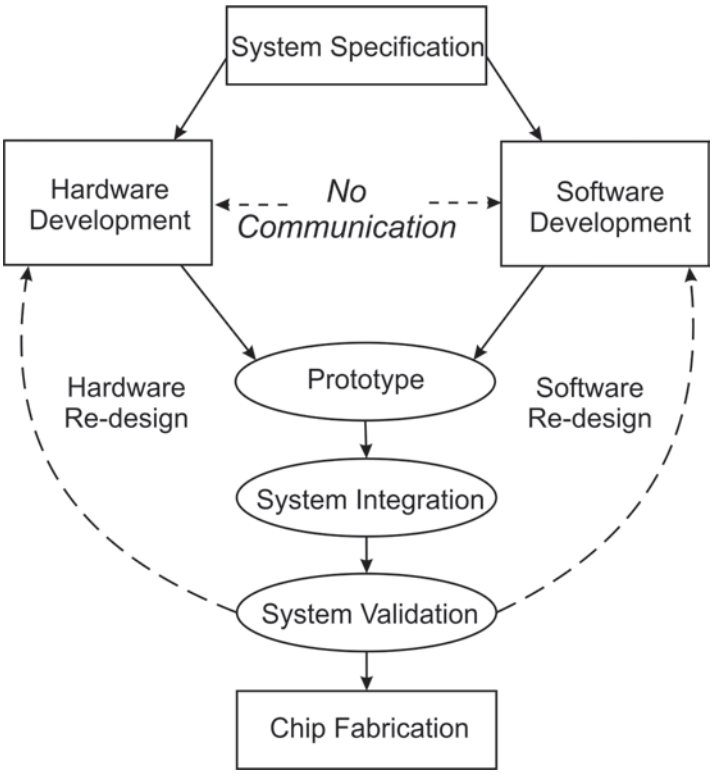


FIGURE 2.3 – *Traditional SoC Design Flow [ed05b]*

level, register-transfer (RT) level, gate level, and transistor level. Each level represents a specific model. We distinguish between the behavioral and the structural model : a behavioral model describes the functionality of the component with different scenarios ; i.e. a graph with different states and the transitions between them. The structural model describes different components of the system and the connections between them.

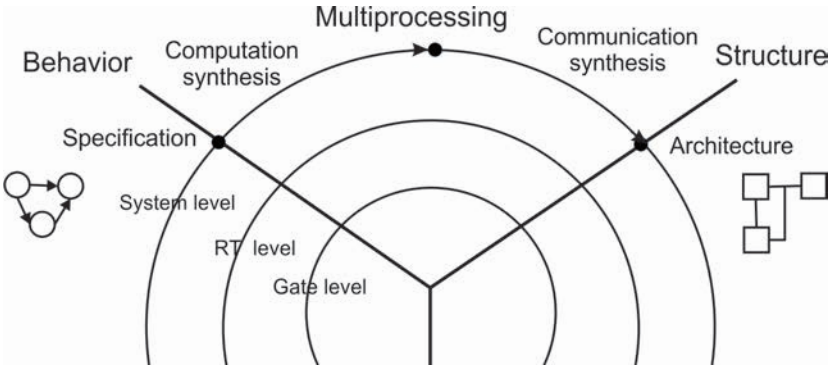


FIGURE 2.4 – *The Y-chart approach for system design [AG02]*

Later, a new methodology for SoC design emerged as presented in Figure2.5. It consists

in hardware/software co-design : first we start from a specification of the system understudy, then the specification is partitioned into the hardware and the software exactly as in the traditional flow design. The main difference with the traditional design process is that both parts are developed and simulated in parallel, which allows the designers to verify the consistency of both parts before final implementation on the SoC. As consequence, designers avoid a long time in the design process.

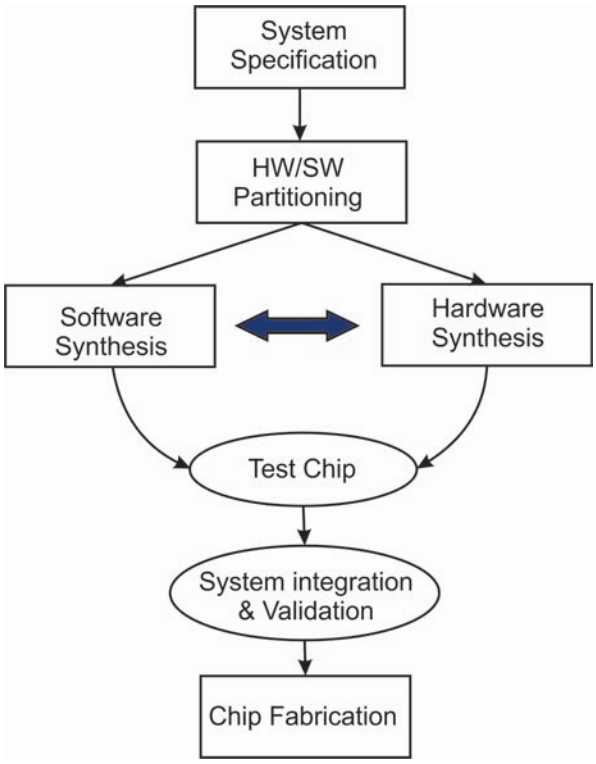


FIGURE 2.5 – *New SoC Design Flow [ed05b]*

CHAPTER 3

Top-Down and Bottom-Up Approaches in the Conceptual Design

Two alternative approaches for the conceptual design were adopted by designers : the *top-down* and the *bottom-up* approaches([VC08, JTB98]). In the top-down approach, we start from the functional description of the design toward solution alternatives, it is also based on the definition of the set of system components from the set of functionalities. Although this approach respects functional requirements of the application but it is not guaranteed that the proposed physical solution is realizable. This is why, this approach is less adopted by engineers since it is time consuming and not efficient in some cases. In contrast, using the bottom-up approach, we start from a pre-defined set of components that we compose together to build the whole system design. Those components are supposed to be functionally correct. In this approach, we ensure that physical realization of components is guaranteed but not the functional requirements. Another problem of this approach is the combinatorial explosion of the design especially in the case of complex systems.

In summary, the *top-down* design defines a set of physical solutions of the application,

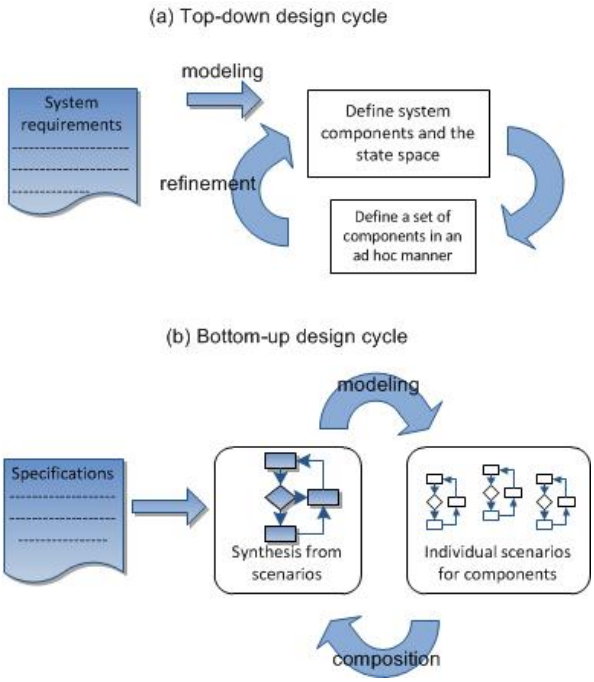


FIGURE 3.1 – *The Top-Down vs the Bottom-Up design cycle[VC08]*

those solutions must respect the functional requirements defined in the beginning of the design process. The *bottom-up* design starts from a description of individual behaviors of the set of the pre-defined components and generates the global behavior of the system under study.

This work describes an integrated conceptual modeling framework that supports the bottom-up approach to give a high level description of SystemC designs using the SystemC waiting-state automata. During this thesis, we take into consideration the limitations of this approach and propose solutions for that.

3.1 The Top-Down Approach

As shown in Figure 3.2, in the top-down approach, we start first by identifying system components with respect to the functional requirements defined in the beginning of the design process. Then, we study the local and global behavior of each component in order to synthesize the functionality of each component. We obtain then a description of a set of communicating processes. This approach requires analysis and refinement during construction of components.

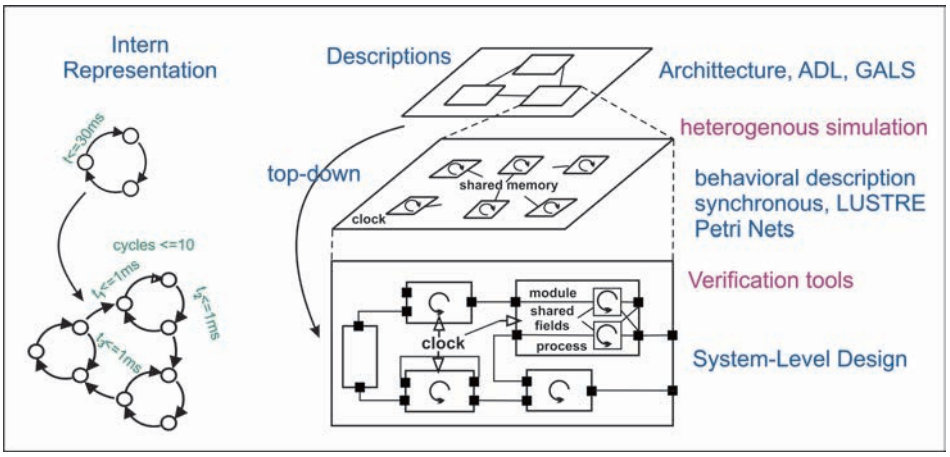


FIGURE 3.2 – *The Top-down approach*

3.2 The Bottom-Up Approach

The bottom-up design methodology is very popular for producing autonomous, salable and adaptable systems often requiring minimal (or no) communication. It has been used to control robotic systems, embedded systems, and sensor networks.

In the bottom-up methodology, components and their environment are modeled in an ad hoc manner. We start from a pre-definition of system components and we study the possible interactions between them : we proceed to composition of individual behaviors of components to build the global behavior of the system. Thus, in this approach, we consider two parameters : *individual behaviors* of components and the interactions between the *components* and their *environment*.

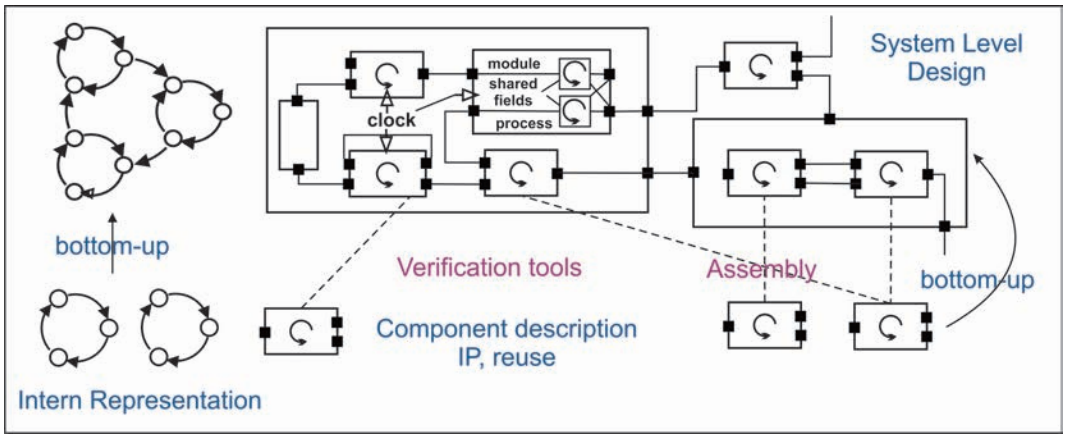


FIGURE 3.3 – *The Bottom-up Approach*

To model the components, we use an automaton for example, which is a transition system where states represent an action that the component is executing. We can resort to different modeling techniques to capture further information about the environment. One can use for example finite state machine (FSM)[AHT06]. Then, we need to define a mathematical representation of the automata in order to validate and analyze the abstract model.

3.3 Conclusion

In this chapter, we studied the importance of SoCs that facilitates the emergence of new technologies but also increases the QoS of new products. Those systems are with a small size but with a bigger performance. Despite this, SoCs become more and more complex due to their embedded heterogeneous components that claim extra and tough work in order to validate the system before implementation. Therefore, the job of engineer designers becomes intricate.

Hence, new methodologies for system design have emerged : we explained here two techniques for system design : the bottom-up and the top-down approaches, we enumerate the advantages and drawbacks of those approaches and we compare them to our approach. We also introduced the traditional and the new systems design approaches and we stressed on the importance of the new design approach for system validation before implementation on the chip. In the next chapter, we present the SystemC language and different levels of abstraction in system design. We stress on the importance of the transaction level (TLM) to reduce the complexity of system verification and validation. We also mention the importance of formal methods to study but also to validate the semantics of SystemC at the TLM.

Part I

System Level Modeling with SystemC

This Part is composed of two Chapters; the first one introduces the transaction-level modeling and the second one describes in details the structural and the syntax of SystemC language.

CHAPTER 4

Transaction-Level Modeling (TLM)

4.1 Introduction

With the ever-increasing complexity of nowadays hardware systems, the job of the designers for the simulation and the validation of those systems has become a tough job : First, simulation is becoming very slow because we need to simulate separately the hardware and the software. Second, it is very difficult to execute different scenarios of simulation and especially detect all errors corners generated during simulation.

Simulation is efficient, in the sense that we can simulate both the hardware and the software but it still be **too slow**. Engineers have to wait for the final chip to write and

execute the program. Consequently, this doesn't respect the time to market pressure nor the development cycle requirements of the design.

Another technique uses the emulators, which are hardware programmable devices, to emulate the behavior of the system on chip. This technique is efficient in term of speed but it still needs an RTL model for the hardware. Later, designers decide to raise the level of abstraction of the design under study in order to model the application with fewer details and to provide an appropriate platform for hardware/software co-simulation. Thus, many attempts to raise the level of abstraction above the RTL have emerged [ed05b].

4.2 Attempts at Raising Abstraction Level

Many attempts were made to raise the abstraction level in order to gain in speed during simulation without loss of accuracy [ed05b]. Thus, the high level model must first simulate the application during a millions of cycles within a reasonable duration of time. Second, the model must be as accurate as possible to give reliable simulation results ; i.e, it must contains enough details about the hardware in order to run the embedded software. Besides, in order to optimize the SOC project cost, the high level model should be developed in a considerable low effort.

First, designers resort to hardware/software co-verification [ed05b]. In SW/HW co-verification, they simulate the software on an RTL model of the hardware and they use a faster processor model that is called Instruction Set Simulator (ISS). It is an instruction-accurate model developed in C language at a higher level of abstraction. The main advantage of that technique is that we can integrate, verify and debug the SOC in an early phase of the design cycle before the implementation of the hardware. The simulation speed of the application is considerably higher in the co-verification of hardware/software. Moreover, any modification on the hardware or the software will be both time and cost efficient since the chip is not manufactured yet. Despite the efficiency of the co-verification in term of time and cost compared to logic simulation, it still lack performance. Thus, it takes a long time to develop the RTL hardware model that is used to simulate the software.

Later, due to the emergence of new modeling languages such as object-oriented languages like C++, Java and later SpecC [DGZ00], in addition to hardware HDLs such as VHDL [vhd02] or Verilog [ver91], designers propose to develop several models. Among them, we

study the *cycle-accurate (CA)* (Figure 4.1) C or C++ models that provide a simulation rate higher than the RTL models in VHDL or Verilog. At this level of abstraction, the model provides an accurate description of both the traditional event-driven simulation and the high level transactions (like bus transactions). It helps software engineers have a vision about what is happening within a clock cycle which is abstracted from system design according to them. It soon becomes obvious that cycle-accurate (CA) modeling has several drawbacks : First, modeling at the cycle accurate level is as complex as the RTL level since the RTL is still a reference for the CA level. The only advantage of CA models is that designers have no constraints about synthesis. Second, there is no gain of speed simulation compared to that at the RTL level ; it was ten times below the original estimations. Third, due to tight scheduling, it is not possible to modify or update the CA model once the RTL model is updated. Thus, the CA model is considered as not fully compatible with the RTL model which is not desired by the modeling engineers.

For all the previous reasons, it is better to model the system at a higher level of abstraction that would allow much quicker modeling than cycle-accurate. This high level model must be precise and fast enough for software developers to test the real embedded software using a standard language enabling reuse of models with a variety of simulator suppliers.

4.3 The Transaction Level Modeling

Transaction-Level Models fill the gap between purely functional descriptions of embedded systems and the RTL description (Figure 4.1). They are created after hardware/software partitioning. TLM is also a transaction-based modeling approach founded on high-level programming languages such as SystemC [sys]. It highlights the concept of separating communication from computation within a system. It also serves as a virtual platform on which the embedded software is executed. The main idea of TLM is to abstract away communication on the buses by so-called transactions : we consider only reading and writing operations on buses. In contrary to the RTL, where everything is synchronized on one or more clocks (synchronous description), TL models do not use clocks. They are asynchronous, they only synchronize through transactions shared between different components. This higher abstraction allows the simulations to be faster than RTL. Figure 4.2 shows an example of compared simulation times for encoding and decoding a picture at the MPEG 4 format [ed05b]. The other advan-

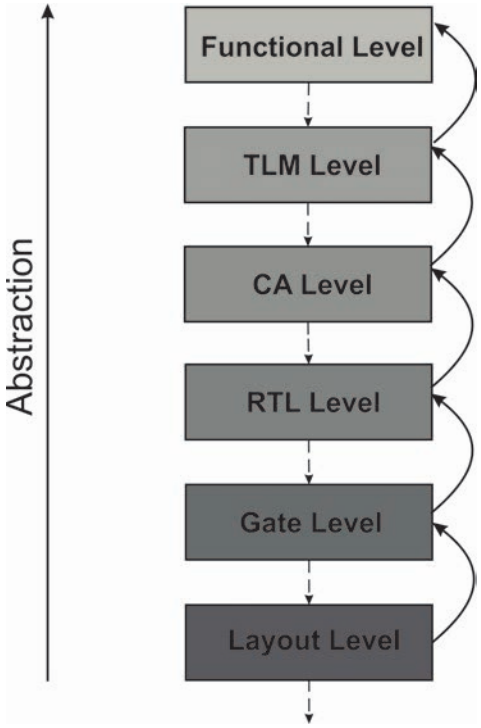


FIGURE 4.1 – *Different abstraction levels for describing the hardware [ed05b].*

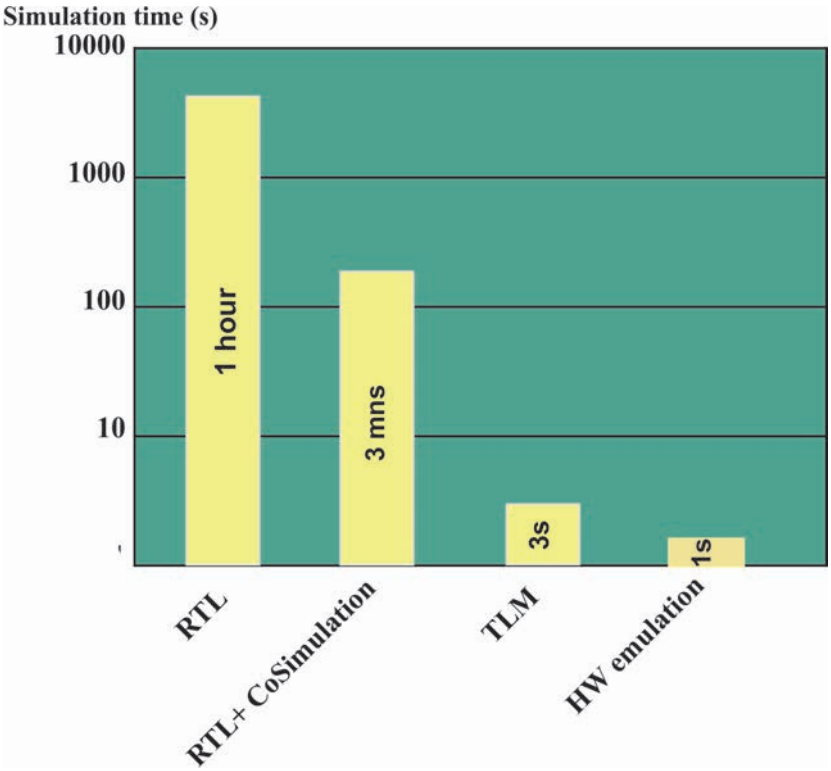


FIGURE 4.2 – *Example MPEG-4 codec (encoder) : Speed in different levels of abstraction (s) [ed05b].*

tage of TL models is that they require far less modeling efforts than RTL or Cycle Accurate models since they are less complex and with less details. Besides, the TL model is completely compatible with the RTL model so we can use it for the hardware validation even after the RTL is created.

4.3.1 Description of TLM

Figure 4.3 shows an example of a TLM platform (model of a TLM Bus). The platform is composed of several components connected through the ports. TLM models each of these components as a **module**. Some of these components may play the role of the communication support such as **channels** : this is the case of the bus model in our example. Components are communicating via transactions, a **transaction** is an atomic data exchange between an **initiator** (or master) and a **target** (or slave). The initiator has the initiative to do the transaction whereas the target is considered as always able to receive it. This corresponds to classical concepts in bus protocols. The initiator issues transactions through an initiator **port**, respectively a target receives them by a target port. Some components only have initiator ports, for instance micro-processors, some have only targets ports (memories). Also, some components contain both initiator and target ports : this is the case for our example (Figure 4.3). Masters receive and send signals via their initiator/target ports.

Synchronization between parallel components is an explicit action between at least two modules (potentially test-benches) that need to coordinate or manage some behavior distributed over them. Such co-operation of different modules is vital to assure the predictable system behavior. TLM is considered as efficient as enough since it provides :

- Early software development ;
- Architecture analysis ;
- Functional verification.

In the perspective of durable progress, TLM leads SoC developers to a number of benefits towards productivity and time-to-market progress.

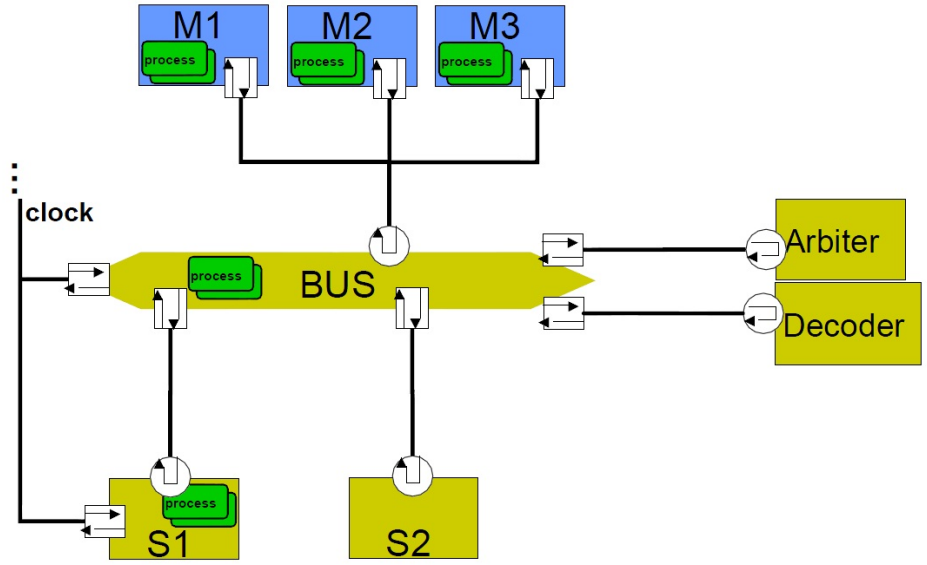


FIGURE 4.3 – *TLM Bus Model (simplified).*

4.3.2 The Modeling Approach with TLM

As discussed earlier, a SoC component is modeled as a module in TLM. A TLM module is a set of hardware blocks or IPs. Each block is represented through its internal functionality, its inputs/outputs and how it is synchronized with other blocks. No details about the architecture or internal pipelines are implemented. A complete SoC TLM platform is constructed by instantiating and binding different modules and channels together. Once the platform is integrated, SoC simulation is performed by executing the related embedded software.

The system synchronization could be modeled by specific means such as events, signals, and interrupts or by data-exchanges. If any of these potential system synchronizations cause a call to the simulation kernel, it enables the scheduler to activate other modules.

We may consider two fundamental classes of TLM[ed05b] :

- Untimed TLM.
- Timed TLM.

On one hand, the untimed TLM is an architectural model targeted specifically at early functional software development and functional verification where timing annotations are not considered or are abstracted from the system description. This model is used to perform simulation speed. It is also called *programmer's view* (PV). On the other hand, the timed

TLM is a micro-architectural model that is annotated with time about the behavioral and communication specifications. The main purpose of timed TLM is to ensure the simulation accuracy that must be respected in real-time embedded software development and architecture analysis. It is also called *programmer's view* plus *timing* (PVT).

Figure 4.4 describes the difference between the untimed and timed TLM with respect to other conventional models in the SoC design flow, which includes register transfer level (RTL), bus cycle accurate (BCA), and cycle accurate (CA) models.

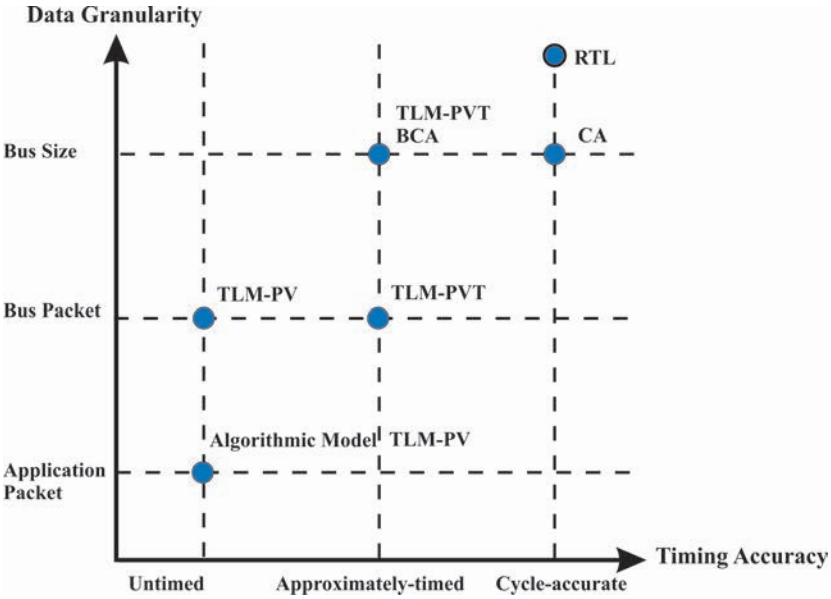


FIGURE 4.4 – *Modeling Accuracy of Various Approaches [ed05b].*

4.3.3 The Novel Design Flow with TLM

Figure 4.5 represents the new methodology for the design flow. It is based on the partitioning of systems into two parts : the hardware and the software. It also describes the position of the TLM in the design flow. It is defined just before the partitioning. Referring to the same figure, a design flow generally starts from user specifications where system requirements are well identified. Based on these specifications, the system is then partitioned into hardware/software parts. But before partitioning, we define the TLM platform that helps both software developers and architectures to develop and simulate their applications. It also helps verification engineers to verify and test the compliance of the RTL platform with the intended performance. They can also verify the hardware and low-level integration of the software part

with the hardware. Once verification is achieved, the chip is finally manufactured.

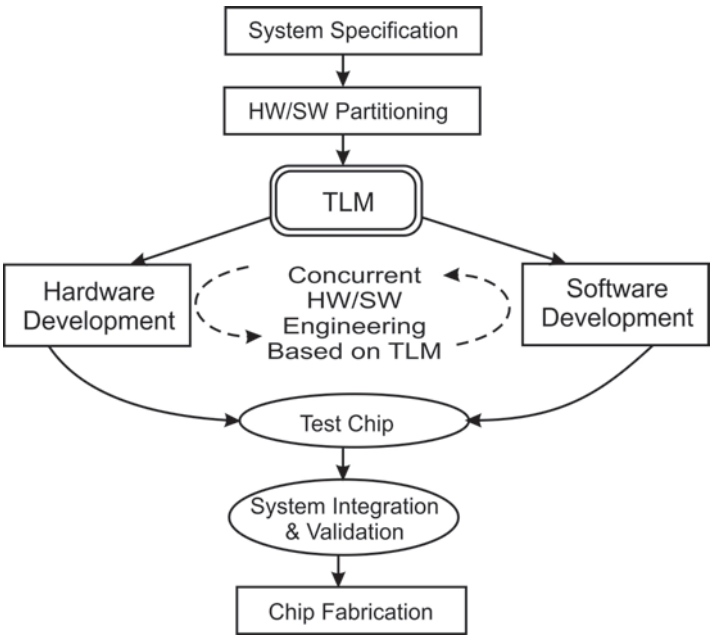


FIGURE 4.5 – *New SoC Design Flow with TLM [ed05b].*

4.4 Conclusion

In this chapter, we define different levels for system design as presented in Figure4.1. We describe the advantages but also the drawbacks of each level. We mention that the job of software designers become easier when using a higher description of the hardware, where the system is described with less and enough details. We then introduce the TLM level and we define its different terminologies and how it improves the new design process.

CHAPTER 5

SystemC Language

5.1 Introduction

SystemC [sys] becomes nowadays a popular language for modeling complex hardware systems. Compared with other hardware description languages, SystemC is more feasible for designing large-scaled complex systems and modeling high level behaviors. It also provides a bridge between hardware and software design and thus, provides a unifying framework for hardware/software design. SystemC consists of C++ libraries and a simulation kernel for

creating behavioral and register-transfer level (RTL) designs. It provides a common development environment needed to support software engineers working in C/C++, and hardware engineers working in HDLs such as VHDL [vhd02], Verilog [ver91], etc., particularly system-on-a-chip designs.

In Figure 5.1, we show the use of SystemC language compared to other programming languages and the reason is clear : increasing design complexity requires very fast executable specifications to validate system concepts, and only C/C++ offers adequate levels of abstraction, hardware/software integration and performance. Besides, nowadays system design demands a single common modeling language that makes the use of new design tools, services and IPs possible.

In response to these needs, SystemC has been developed as a standardized modeling language intended to enable system level design and IP exchange at multiple abstraction levels, for systems containing both hardware and software components.

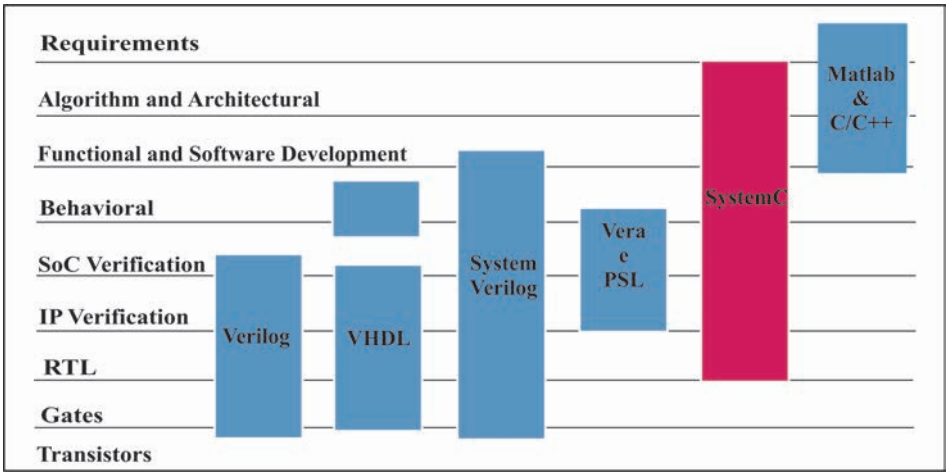


FIGURE 5.1 – *Levels cover by different programming languages.*

5.2 Structure of a SystemC Model

The SystemC is a System-Level Modeling language based on C++ that is intended to enable system level design in response to the need of a very fast executable specification to validate and verify system concepts.

Using the SystemC library, a system can be specified at various levels of abstraction. For hardware implementation, models can be written either in a functional style or in a

register-transfer level style. The software part of a system can be naturally described in C or C++. SystemC uses the object-oriented (OO) approach to achieve abstraction, modularity, compositionality, and reuse. The OO paradigm in SystemC is not incidental but central. It distinguish SystemC from other modeling languages, such as SpecC. The base layer of SystemC provides an event-driven simulation kernel. This kernel operates at the event level and switches execution between processes.

5.2.1 Syntax

For simplicity, we omit the syntactic elements for representing the architecture of a SystemC program as mentioned in Table 5.1. It adopts a C-like syntax :

Program	{modules, channels, signals, events, variables}
Module	{ports, variables, process-decl, process-body, methods}
Process-decl	$< processname >< sensitivity >< reset - condition >$
Process-body	$< event - comm \mid signal - comm \mid chan - comm \mid control - flow \mid arithmetic >$
Event-comm	$wait(event)$, $wait(event, time)$, $wait(time)$, $wait(eventlist)$, $notify(event)$, $notify - delayed(event)$
Signal-comm	$signal.read \mid signal.write$
Chan-comm	$tlm\_port \rightarrow put(value) \mid tlm\_port \rightarrow get(var) \mid$ $tlm\_port \rightarrow method(parameters)$
Control-flow	$< C++ control flow >$
Arithmetic	$< C++ arithmetic >$

TABLE 5.1 – Simplified abstract syntax for SystemC.

Syntactically, a SystemC program consists of a set of modules(Figure 5.2), a module contains one or more processes to describe the parallel aspect of the design. a module can also contain other modules, representing the hierarchical nature of the design. Processes inside a module are communicating via signals. Modules communicate via channels. Channels are abstract and are accessed via their interface methods. The simulation kernel, together with modules, ports, processes, events, channels, and interfaces constitute core language of C++. That is accompagned by a collection of data types. Over this core, SystemC provides many library-defined elementary channels, such as signals, FIFOs, semaphore, and Mutex. On top of this are defined more sophisticated libraries, including master/slave library, and process networks. A transaction-level modeling library (TLM 1.0) was announced in 2005. SystemC has been developed with heavy intermodule communication in mind. SystemC 1.0 [sys] pro-

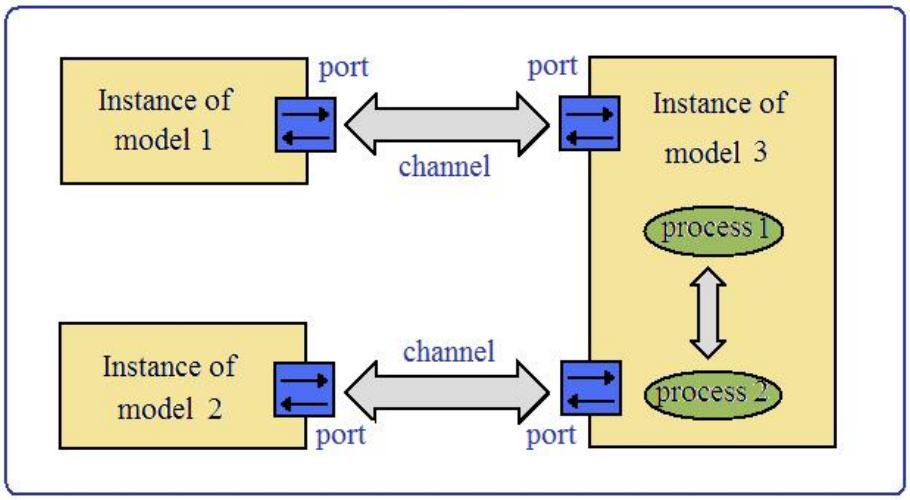


FIGURE 5.2 – *Modeling in SystemC.*

vides structural description features including modules and ports that can be used in systems design. In addition, there exist different data types to enable modeling hardware systems and processes to express concurrency. SystemC 2.0 [sys02, sys05] introduces channels, interfaces, and events to enable communication and synchronization between modules or processes. An interface specifies a set of access methods to be implemented within a channel, where channels provide the implementation for these interfaces. An event is a flexible synchronization primitive that is used to construct other forms of synchronization. Different channel types are defined with respect to some rules. SystemC imposes rules on channels and the way they communicate. Those rules include how many ports are connected and what the interface types that these ports require. On the other hand, dynamic design rules checking is needed to ensure that channels do not violate these rules during simulation time.

Most HDLs, VHDL for example, use a simulation kernel. The purpose of the kernel is to ensure that parallel activities (concurrency) are modeled correctly. The behavior of the simulation should not depend on the order in which the processes are executed at each step in simulation time. The SystemC simulation kernel supports the concept of delta cycles. A delta cycle consists of an evaluation phase and an update phase. This is typically used for modeling primitive channels that cannot change instantaneously. By separating the two phases of evaluation and update, it is possible to guarantee deterministic behaviors.

To conclude, SystemC semantics combine then the semantics of C++ with the simulation

semantics of the kernel. The simulation semantics are event driven rather than cycle driven. But at the same time, SystemC has a discrete model of time, which means that it has also cycle-level semantics.

5.2.2 Processes

Processes (Figure 5.3) inside a module are of three types : *Method*, *Thread* and *Clocked Thread*. However, methods and clocked threads can be modeled as threads without loss of generality. Similar to VHDL or Verilog, a process has a list of events that activate it. This list of events is called the sensitivity list of the process. As soon as the event occurs, the process is activated and executes until the process terminates or suspends its execution by means of the **wait()** statement. The SystemC methods are special cases of processes that do not call **wait()**. Events may either be generated explicitly by a thread (using the **notify()** statement or method), or implicitly by changing signal values. SystemC specification distinguishes three states of a thread : *running*, *waiting* and *runnable* (as mentioned in Figure 5.4). A running thread may generate events that activate other threads sensitive to those events and change their states to runnable. A single event may trigger the execution of multiple threads. A running thread may become a waiting thread by executing the wait statement. The scheduler chooses a thread among the runnable threads to resume execution. As in Verilog, the ordering in which the runnable threads are activated is chosen non-deterministically. It is important to note that no interleavings are done between the threads unless a **wait()** statement is executed. It is important to note that the synchronization does not happen upon the generation of the event, but only upon calling wait().

5.2.3 Channels

Communication between processes can also be accomplished through channels. A channel can be regarded as that it consists of two buffers, one for storing its *current* value and the other for storing its *new* value. Each execution of a channel output statement generates a request to update the channel if the value of the expression is different from the current value of the channel. The pending requests will be carried out in the following update phase. If more than one channel output statement to the same channel occur during a particular evaluation phase, the last one executed determines the new value of the channel in the following update phase.

```
SC_MODULE(my_module) {
// input port
// output port
void my_process ( );
void my_thread( );
...
SC_CTOR(my_module) { // Constructor
SC_THREAD(my_thread); // Thread Process
// make thread sensitive to change of input
sensitive << .. << .. ;

SC_METHOD(my_method); // Method Process
// make thread sensitive to change of input
sensitive << .. << .. ;
}

\\Structure of the Method and the Thread processes

SC_METHOD : simulation engine call them repeatedly
void my_method ( ){ // run to completion scheme
// treatment
}
SC_THREAD :
void my_thread( ){ // infinite loop scheme
while (1){
// treatment
wait(); // static event
}
}
```

FIGURE 5.3 – *Structure of a module.*

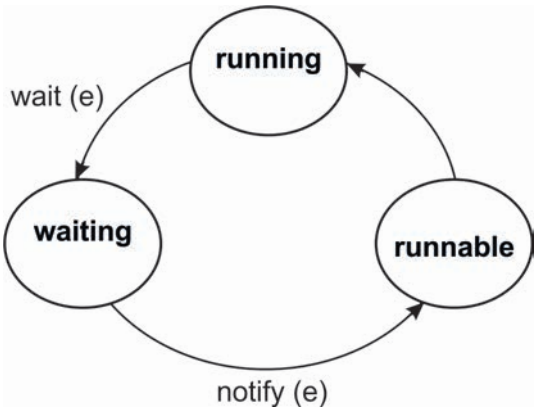


FIGURE 5.4 – *Transitions between the states of a thread.*

Either in simulation or in the real world, hardware signals do not immediately change their output value when they are assigned a new value. The concept of delayed channel assignments plus delta-cycle provides the ability to properly model hardware signals. A delta-cycle can be thought of as a very small step of time within the simulation, which does not increase the user-visible time.

5.2.4 Events

Despite of the diversity in syntax, SystemC is essentially an event-driven model and all communications in SystemC models are implemented using events and the associated wait/-notify mechanism. An event is a flexible, lowlevel synchronization primitive that is used to construct other forms of communication.
Events can be used only with certain constructs such as *wait* and *notify* :

- The *wait* statement suspends the execution of the current thread waiting for one or more events to occur.
- The *notify* statement generates the events specified as arguments for some threads. The execution for all threads that are waiting for these events is resumed.

The occurrence of an event may activate processes that are *waiting* for it. According to the way events are notified, there are three kinds of event notifications : *immediate* notifications, *delta-cycle delayed* notifications and *timed* notifications. Delayed event notifications are widely used in modeling hardware behaviors and software systems while immediate event notifications are useful for modeling software systems and operating systems, which lack the concept of delta-cycle.

SystemC events can be roughly classified into the following three sorts :

- *User-defined events* : These are events defined by SystemC programmers in source code. Such events are usually triggered by the command *notify* ;
- *Channel events* : These are pre-defined SystemC events and they are triggered when something occurs on channels. For instance, an event denoting the arrival of a new value will be triggered whenever some value is written to a sc_buffer channel. Channel events at different types of channels have different semantics ;

- *Clock events* : Clock signals are also seen as events and they are usually defined as *sc_clock* in the SystemC main program. SystemC core engine is in charge of generating sc_clock events at proper time. We never notify a clock event in the program.

We shall not distinguish between the first two sorts of events since both of them can be dynamically notified. Clock events are rather seen as the input events of SystemC models. In fact, we prefer not considering any pre-defined channels or signals in our modeling, but rather taking into account their event-driven implementation.

According to the way events are notified, there are three kinds of event notifications : *immediate* notifications, *delta-cycle delayed* notifications and *timed* notifications. Delayed event (delta-cycle, timed) notifications are widely used in modeling hardware behaviors and software systems while immediate event notifications are useful for modeling software systems and operating systems, which lack the concept of delta-cycle.

- Immediate notification of an event e : it is achieved with *e.notify()* which means that event e is triggered in the current evaluation phase of the current delta-cycle. Any processes that are waiting for the event e will be made ready to run. Immediate event notifications cannot be canceled since their effect occurs immediately.

- Delta-cycle delayed notification of an event e : it is achieved with $e.notify(\Delta)$ which means that event e will be triggered after the current delta-cycle. This delayed notification can be canceled by executing statement *cancel(e)* before it is triggered.

- Timed notification : it is achieved with *e.notify(t)* which means that event e will be triggered after a period of specified simulation time t. This delayed notification can be canceled by executing statement *cancel(e)* before it is triggered.

The effect of statement *cancel(e)* is to cancel all the delayed notifications on event e. For any given event, at most one pending notification can exist and statement cancel only cancels pending notifications. An event has only one pending notification. More than one notification on the same event are resolved according to the following rule : an earlier notification will always override the one scheduled to occur later. An immediate notification is taken to occur earlier than a delta-cycle delayed notification while a delta-cycle delayed notification occurs earlier than a timed notification. This is irrespective of the execution order of statement *notify*.

5.2.5 Time in SystemC

Besides events, a process may wait for a period of time, either a delta-cycle or a period of specified simulation time. In these cases, we say the process waits for a timeout. There are two kinds of timeouts. *Delta-cycle* timeout stands for delta-cycle advancing and *simulation time timeout* stands for simulation time advancing. A process may also wait for some events and a simulation time timeout simultaneously, where any occurrence of the two parts resumes the process.

5.3 SystemC Scheduler

The simulation of SystemC models is managed by the SystemC scheduler, which can be seen as a total event-driven model : communications through ports and channels, clocks, and actions of modules, are all triggered by (different types of) events. The basic unit of the simulation is the so-called delta-cycle and a complete simulation procedure is just a sequence of delta-cycles. The scheduler maintains several tables, among which we are particularly interested in the table of *runnable processes* (processes that are ready to execute at the current delta-cycle). Here is a brief description of a delta-cycle : a delta-cycle starts with a non-empty runnable process table. The scheduler executes these processes one by one, in a pre-defined order ; every runnable process executes until it ends or it is pended again (by a wait command for instance) ; if any immediate event is notified during the execution of a runnable processes, it will add processes that are currently sensitive to this event into the runnable process table ; delta-events and timed events that are generated during the execution of a process are stored in other tables. The process table is emptied when all runnable processes are executed and the procedure of executing all the runnable processes is called a *evaluation phase*. The scheduler then checks those delta-events notified in the evaluation phase : if there are processes that are sensitive to these events, then add them into the process table. This procedure is called a *delta notification phase*. If the process table is non-empty, the scheduler enters the next delta-cycle and executes the evaluation phase again ; otherwise, it checks the timed events notified in the evaluation phase and adds processes that sensitive to these timed events into the process table. This is called a *timed notification phase*. The scheduler then advances the simulation clock and enters the next delta-cycle. A detailed explanation and

implementation of delta-cycles can be found in SystemC documents [sys]. However, for the sake of clarification, we prefer regarding a delta-cycle as starting from a delta notification phase or a timed notification phase. In other words, a delta-cycle in this paper will start with a set of events which add all sensitive processes into the process table, then continue with an evaluation phase.

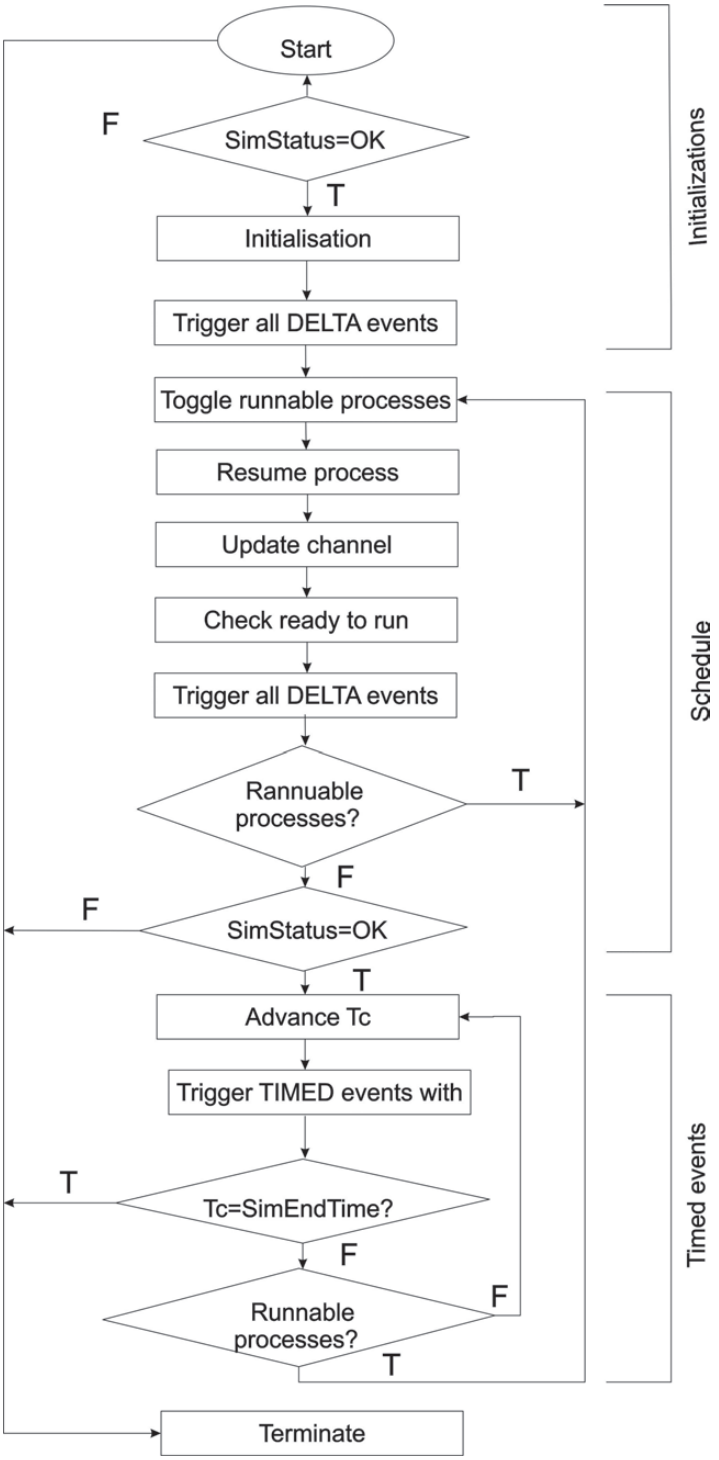


FIGURE 5.5 – *Execution semantics of SystemC*

5.4 The TLM Library

In the transaction modeling (Figure 5.6), the system is divided into two parts : the *communication* part and the *computation* part. In this definition, TLM is considered as modeling the communication part of a system at a high level of abstraction (e.g., by functions). While, the computation part (modules) of a design can be at various levels of abstraction. It is obvious that at the higher level the modules are designed, the faster their simulation process and the easier their connection with the communication part are. The TLM library defines

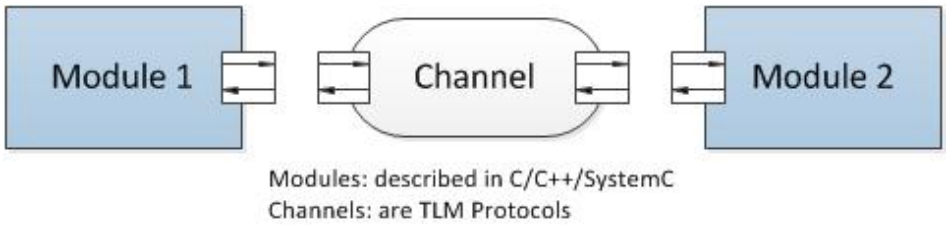


FIGURE 5.6 – *TLM Mechanisms*

several abstract, transaction-level interfaces and the ports and exports that facilitate their use. Each TLM interface consists of one or more methods used to transport data, typically whole transactions (objects) at a time. Component designs that use TLM ports and exports to communicate are inherently more reusable, interoperable, and modular. Processes in TLM interface can be declared *blocking* when they suspend their execution by calling *wait()* for example or *non-blocking* like *SC_METHODs* that can not be suspended during execution until they achieve.

Transactions between modules can be *bidirectional*, for example a read across a bus. they can be also *unidirectional*, as it is the case for most packet based communication mechanisms. Where there is a more complicated protocol, it is always possible to break it down into a sequence of bidirectional or unidirectional transfers. For example, a complex bus with address, control and data phases may look like a simple bidirectional read/write bus at a high level of abstraction, but more like a sequence of pipelined unidirectional transfers at a more detailed level. Any TLM standard must have both bidirectional and unidirectional interfaces. The standard should have a common look for bidirectional and unidirectional interfaces, and it should be clearly shown how to relate both of them.

Part II

Modeling with SystemC Waiting State Automata (WSA)

This Part is composed of five Chapters ; it represents a detailed description of the SystemC waiting-state automaton model.

CHAPTER 6

Introduction and Related Works

In Part I, we presented the SystemC language as a system-level language used to model complex systems. The hardware and the software parts are represented at different levels of abstraction : the software can be naturally described in C or C++ language, the hardware is described either using the transaction level or the register-transfer level (RTL) models. Information about the functional as well as the non-functional properties of the sytem can be added to the description of the system in order to refine the final model. Hence, SystemC provides a complete framework to help developers to easily model complex hardware systems at different levels of abstraction.

We also presented the transaction-level (TLM) as an early level for hardware design on which the embedded software can be run. We stressed on the advantages of this level in the conceptual design : first, because it fills the gap between the purely functional description of the system and the RTL description since it allows hardware/software co-design. Second, because it separates the communication part from the computation part within a system,

which makes the job of the conceptual designers more easier. It also abstracts away the communication on buses by transactions : i.e, reading and writing operations. Besides, models in TLM are supposed to be asynchronous because they don't synchronize through clocks like in RTL but rather through transactions.

6.1 Introduction

As we mentioned in the previous chapter, SystemC provides an efficient framework to model and co-simulate hardware/software systems before the final implementation on the chip. Throughout the conceptual design, designers can verify in parallel the hardware as well as the software description of the system and any modification on the hardware or the software can be detected and easily applied in earlier stages of the conceptual design. Despite this, it is not usually easy to detect most errors in the conceptual design particularly in critical parts of the system implementation. Therefore, there is a need to resort to formal techniques that help the designers to detect errors in the early stages of the software development. Over the last years, research activities were mainly focused on exploiting modeling flexibility and exploring different levels of communication and behavior abstraction. More recent works concentrate on the formalization and the verification of SystemC. In this chapter, we enumerate the main and recent approaches in SystemC modeling.

6.2 Existing Static Approaches in SystemC Modeling and Verification

Große and Drechsler [De02, De03] focused on the verification of SystemC at the gate level. Their work consists in verifying properties of synchronous sequential circuits using the LTL (*Linear Temporal Logic*) [Pnu77]) which is a modal temporal logic with modalities referring to time. In LTL, one can encode formula about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, etc. The formula LTL is then translated into a decision problem (SAT) using the Binary Decision Diagram (BDD)[Ran92]. The main drawback of this approach is that it was somehow limited to the gate level and doesn't support the transaction level. For example, if we want to verify some critical properties like *vivacity*, this may lead to prohibitive computation. But later in [DGeD10],

they propose a fully automatic approach to verify SystemC properties at the transaction level using C assertions and finite state machines.

Mueller et al. [WMR03] translate SystemC program simulation using Abstract State Machines (ASMs)[BS03]. The ASMs have been extensively used in the definition of different hardware modeling and description languages, but it is still not efficient since it does not capture the synchronization between processes at the waiting states. Besides, their model is not adapted for new techniques for programs analysis like model checking or abstract interpretation. Later Gawanmeh et al. [AGT04], use also ASM to model SystemC designs. They use AsmL modeling language of ASM to define the semantics of SystemC language at the transaction level. They define the semantics of the SystemC simulator as well as non-trivial SystemC components including FIFO channels, MUTEX channels, message queuing, request-grant protocol and SystemC FIFO hierarchical channels with handshake protocol. Their approach is also interesting but not efficient, first because it doesn't capture all SystemC components and second, because it doesn't stress on the synchronization between concurrent processes. Besides, in their semantics they need to model separately the behavior of SystemC scheduler and thus they have to define in advance the scheduling policy.

Kroening et al. [KS05], they represent SystemC models using the Labeled Kripke Structures (LKSs) [MCBG88]. In fact, the labeled kripke structure model is based on the state/event analysis and it makes use of the formalization of labeled Kripke structures. Having separated labels on states and transitions in the model provides a syntactic way of partitioning a SystemC model into a hardware and a software part. States in their model include all possible intermediate states within a process, they also make a classification of processes as runnable processes, waiting processes, etc., which is basically the implementation idea of SystemC scheduler. On the other side, their labels on states allow effective manipulation of program data. In conclusion, their approach proposes a system level representation of systems by automatically partitioning the uniform system description into synchronous (hardware) and asynchronous (software) parts.

Habibi et Tahar [AHT06], they translate the logic properties and the UML behavior of SystemC codes into Abstract State Machines using the AsmL language (a specification executable

language developed by Microsoft). This model is then used to generate Finite State Machines (FSMs), they propose two algorithms to generate *finite state machines* from SystemC models. These generation algorithms are used for traditional model checking adapted for SystemC. Their model generates the states for the whole system from the very beginning. Then the algorithms focus on solving the state exploration problem using the grouping technique. the FSM serves as a precise model of the observable behavior of the system used to validate lower abstraction levels of the design (the register transfer level (RTL)).

Karlsson et al. [K06], their approach is similar to the work of [LACP04]. They propose a formal representation of SystemC models at a high level of abstraction (TLM) using Petri-nets that can be used for model checking of properties expressed in a timed temporal logic.
Although this approach is efficient to represent SystemC parallel designs in a formal and efficient way but it still be inadequate for complex systems where interactions are intricate. In fact, modeling complex systems with petri-nets require to consider and represent all possible interactions between concurrent components and communication between them. This may lead to state explosion of the final model of the system, which should be avoided while modeling embedded complex systems. To understand the modeling process using petri-nets representation, we consider the following petrinet example which models the program above. Authors in [K06] use a design representation called Petri-net based Representation for Embedded Systems (PRES+). Each SystemC statement is represented by one place and one transition. The transition performs the actual statement, whereas a token in the place enables the execution of the statement. Variables are also represented by places.
In this example, we need 8 places to just represent a 4 lines program. If we consider a program with hundreds of lines and variables, the representation becomes more and more complicated to manage. This is the main drawback of this approach. Another disadvantage of this approach is that it does not support verification of properties like concurrency and interactions between processes at the delta-cycle level which is one of the main features to study in SystemC language, since it represents the simulation policy of SystemC scheduler.

Moy et al. [MFM06], they use the LusSy tool to extract information about the system architecture and behavior in the transaction level. They also use abstraction on their design to build an intermediate model, that they call HPIOM (*Heterogeneous Parallel Input/Output*

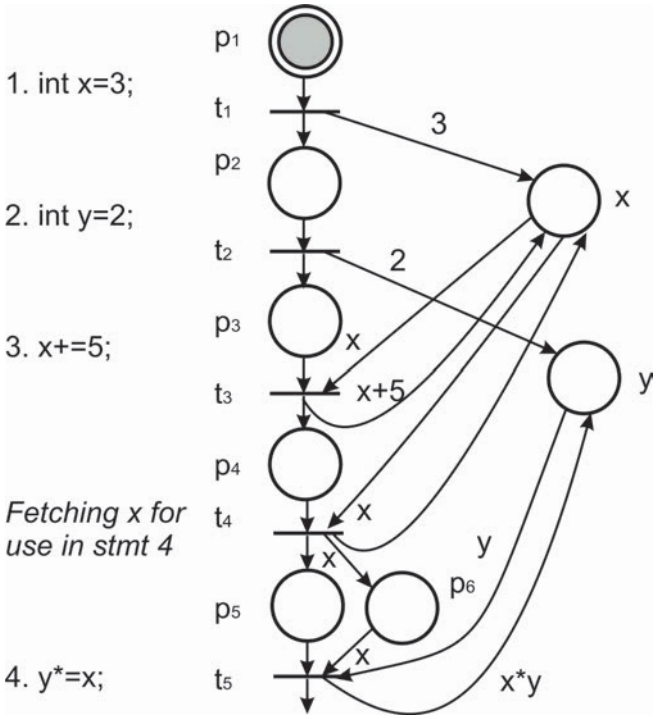


FIGURE 6.1 – *A example of petri net representation (source [K06]).*

Machines). They can apply model checking techniques to verify the HPIOM models (SMV [smv] and spin [spi] for example). Their global approach consists in extracting an automaton for each process and an automaton for each TLM SystemC component. They use abstraction techniques to avoid state explosion on HPIOM automata. Their approach captures most SystemC semantics, also it is amenable for new model checking techniques. The main drawback of this approach is first the state explosion problem since they have to model all components states which should be avoided in systems modeling. Besides, they need to model separately the behavior of the scheduler using a HPIOM automaton. Moreover, their approach doesn't benefit of powerful software verification techniques like predicate abstraction [GS97].

Blanc et al. [NB08], authors propose a new tool called SCOOT which help extract and optimize a formal representation of SystemC programs at low levels. Authors also suggest the possibility to re-synthesize C++ code from the model built using the SCOOT tool which may performs simulation results. They propose to use *model checking* techniques to verify the resulted model like in [KS05]. To guarantee accurate results during verification, authors partition the system into two parts : the software part and the hardware part. They distinguish

between three types of processes :

- *Combinatorial Processes* : their activation depends on the entries and not the clock, they don't have a **wait** statement and no unlimited loops ;
- *Processes with clocks* : they depend on clocks and they have no unlimited loops ;
- *Processes with restrictions* : they represent the software part.

The combinatorial process is transformed into mathematical expression and then reduced from the model. Processes without restrictions are unchanged. Only processes with clocks represent the hardware part, they are modeled using state machines by using the Kripke structure [ECS04]. Each component is then presented by a state machine.

6.3 Summary

In the following table, we study a comparison between the previous approaches : we resume briefly the description of each approach. Then, we precise the abstraction level handled by the approach and finally we conclude by the limitations of each approach.

The Approach	The abstraction level	A brief description	Limitations of the approach
Grosse and Dreschler	Gate level	- Verify the properties of synchronous circuits using LTL . - The LTL formula is then translated into BDD.	- It is limited to low levels
Muller et al.	TLM	Translate SystemC into Abstract State Machines	- It doesn't handle all SystemC semantics. - They also need to define in advance the scheduling policy.
Habibi and Tahar	TLM	- Translate the logic properties and the UML behavior of SystemC codes into Abstract State Machines	- Difficulty in state exploration due to their composition algorithms.
Kroening et al.	From RTL to TLM	- Represent SystemC models using the Labeled Kripke Structures (LKSs) by hardware/software partitioning.	- State explosion.
Karlsson et al.	TLM	- They propose a formal representation of SystemC models using Petrinets. - The properties are expressed in a timed temporal logic.	- State explosion.
Moy et al.	TLM	- They propose a formal representation of SystemC models using Heterogeneous Parallel Input/Output Machines.	- State explosion. - Model separately the behavior of the scheduler. - The approach doesn't benet of powerful software verication techniques like predicate abstraction.
Blanc et al.	Low levels	- They propose a new tool called SCOOT to extract and optimize a formal representation of SystemC programs.	- It doesn't handle high level representation of SystemC.

CHAPTER 7

SystemC Waiting State Automata

In this chapter, we will introduce a formal model for SystemC modeling : the SystemC waiting-state automata. The SystemC WSA is based on the analysis of the $wait/notify$ mechanism of SystemC which plays an important role in the SystemC scheduler. Modeling SystemC designs using automata can be suitable to model parallelism between different components which is essential for hardware description. This choice will be different if we have distributed systems where a few heterogeneous components communicate in parallel or for sequential processes. Although, Petri-nets for example are considered to be more appropriate to handle parallelism, they still have a considerable problem that is the combinatorial explosion of the system states which is significantly reduced in the SystemC waiting-state automata. Besides, other drawbacks of peti-nets are : first, they don't handle the representation of SystemC designs at different levels of abstraction, more precisely the delta-cycle level, although,

they are efficient to represent SystemC designs at the system-level. Second, petri-nets do not allow expressing timing properties and counters, that represent the system evolution, like in the SystemC waiting-state automata.

The original model was first proposed by Zhang, Védrine and Monsuez in [YZM07] and later was extended and developed by Harrath and Monsuez in [HM09, HM12]. In the original paper [YZM07], authors reveal the main idea behind the abstract model. They also define its formal syntax and illustrate it on the FIFO example. Later in [HM09, HM12], we extend the model with timing parameters and we illustrate how to use the timed model to verify temporal properties of SystemC designs. It was proved in [HM09, HM12] that this model is compositional since it guarantees that possible interactions between the SystemC process and its environment is already taken into account. In [HM12], it was mentioned that the model is conform to the low-level simulation semantics of SystemC. Besides, it detects anomalous behaviors generated due to concurrent access to shared resources. In [HM12], the model is compared to existing approaches that study and model the SystemC language, which was not expressed in [YZM07].

As for applying model checking techniques [ECP99a] in later stages of system verification, it is essential to define an internal finite representation for SystemC designs using state-transition systems. This representation is amenable to verify additional properties of modules : structural properties (liveness and determinism), properties related to the QoS (quality of service), as well as functional and non-functional properties of embedded systems.

7.1 Motivations and General Approach

In this thesis, we adopt an internal *bottom-up* approach based on SystemC waiting-state automata (WSA) as presented in [YZM07]. In opposition to the *top-down* approach (Chapter 3) : the approach starts from a low level representation of SystemC components and then it assembles all components in order to build a global model for the whole system. But before assembling all components together, it is mandatory to ensure that each component satisfies specific constraints and that it is able to gradually introduce the concepts of quality of service from more functional concepts.

There are many motivations behind the idea of using the SystemC waiting-state automata to represent SystemC components : first, it is essential to give an internal representation of

each system component using a state-transition system since it is easier to verify properties on single components rather than on the whole system. Second, giving a finite representation of an infinte system, which is the purpose of the SystemC WSA, is one of the main goals of most researches for nowadays system modeling. Thus, some existing works try to apply new abstract techniques to give a finite representation of embedded systems, which is already applied on the SystemC WSA semantics. Besides and more particularly, it was mentioned and proved in [YZM07, HM09, HM12] that SystemC WSA is conform to SystemC simulation semantics since it represents the behavior of the system components within a delta cycle : the smallest simulation unit of time of SystemC scheduler. In addition to that the model represents the system at different levels of abstraction, as we will prove later, starting from system level modeling to the delta-cycle level modeling. Another important point is to separate the internal behavior from the global behavior of each component which is essential when modeling parallel systems. Thus, in SystemC WSA, authors consider only states where components are communicating with the environment waiting for the notification of some events in order to resume execution. Accordingly, the internal states that represent local behaviors of each component are abstracted from the system representation.

Unlike other formal models used to verify SystemC designs such as in [AHT06, KS05, MFM06, KMS06], the SystemC waiting-state automaton is different since it considers only interactions and communications between processes and the way they are managed by the SystemC scheduler. It supposes that the behavior of a process between two wait states is abstracted in the resulted model. The SystemC waiting-state automaton stresses on two main concepts :

- ❒ The set of the entry-conditions which activate and suspend the execution of a process and the set of the exit-conditions that are generated.

- ❒ The synchronizing points in the SystemC program used to synchronize between the communicating processes within a delta cycle.

The idea behind the SystemC WSA is to build an automaton for each process. The automaton is build from the set of the waiting-states, so it is considered as an abstraction or a minimal representation of the initial program. This is why, we call each automaton as the **minimal-step automaton**, we will use this notation throughout this thesis.

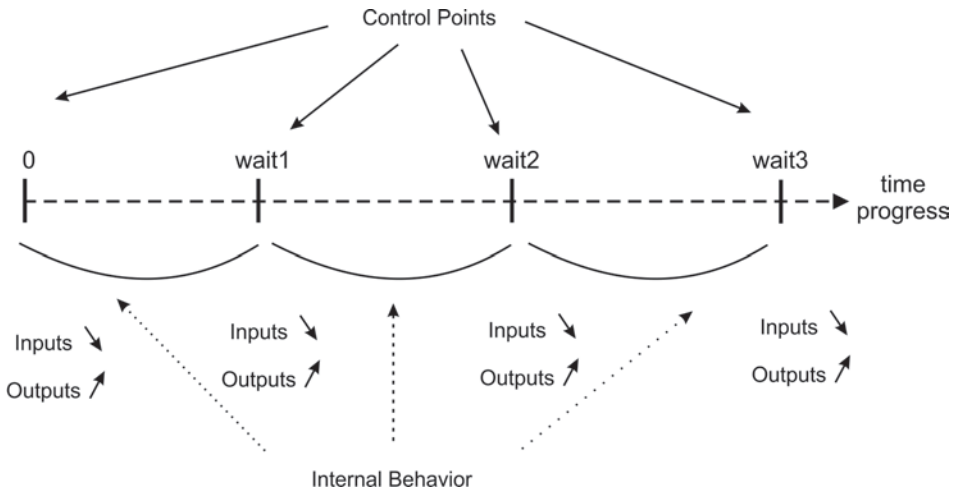


FIGURE 7.1 – *The execution semantics of SystemC.*

Let's start from a brief description of the execution semantics of SystemC (Figure 7.1) : at the start of each time step, inputs to the program are obtained from the environment, all computation is viewed as instantaneous (i.e., occurring in zero time). There is one special statement *wait()*, that affects time advancement. When a *wait* statement is encountered, any changes to the program's outputs become visible to the environment, this is the step when time obviously progresses. Thus, computation proceeds as follows : Obtain inputs, compute (in zero time) until a wait is encountered, make output changes visible, obtain new inputs, etc. The *wait* statements represent the *control points* in the program, i.e, processes can only *suspend* and *resume* execution when they call *wait()* statements. So SystemC models can be written without concern that a process may be pre-empted involuntarily. Specially, the code within a process delimited by two wait() statements can safely assume that no other processes have modified any variables which are also accessible to other processes and the execution of the code occurs instantaneously.

7.2 Example

Let us start by a simple SystemC model : an implementation of FIFO with clocks. The structure of the model is shown in Figure 7.2.

The implementation of the modules in SystemC is given in Figure 7.3 and the main program is given in Figure 7.4.

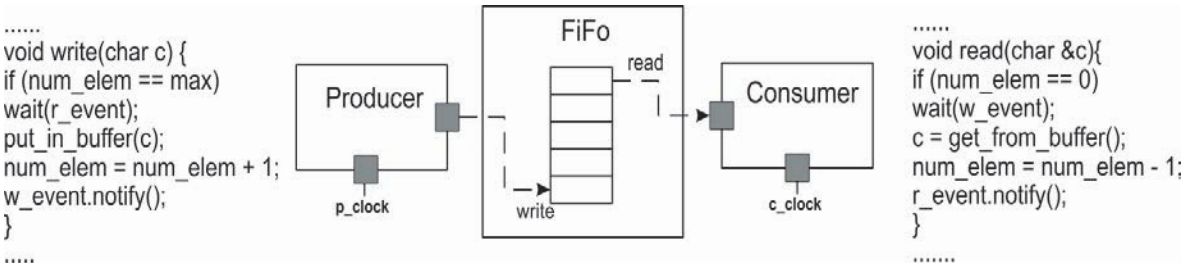


FIGURE 7.2 – *The FiFo example.*

```
class fifo_if : virtual public sc_interface{
 public:
  virtual int write(char) = 0;
  virtual int read(char &) = 0;
  virtual int num_available() = 0;
};
class fifo :
public sc_channel, public fifo_if {
 private:
  enum e { max = 2 };
  char buffer[max];
  int num_elem;
 public:
  sc_event w_event, r_event;
  fifo(sc_module_name name) :
    sc_channel(name), num_elem(0) {}
  void write(char c) {
    if (num_elem == max)
      wait(r_event);
    put_in_buffer(c);
    num_elem = num_elem + 1;
    w_event.notify();
  }
  void read(char &c){
    if (num_elem == 0)
      wait(w_event);
    c = get_from_buffer();
    num_elem = num_elem - 1;
    r_event.notify();
  }
  int num_available() { return num_elem;}
};
```

```
class producer : public sc_module {
 public:
  sc_port<fifo_if> fifo;
  sc_in_clk p_clock;
  SC_HAS_PROCESS(producer);
  SC_MODULE(producer) {
    SC_THREAD(main);
    sensitive_pos << p_clock;
  }
  void main() {
    while(true) {
      wait();
      produce(c);
      fifo->write(c);
    }}};
```

```
class consumer : public sc_module {
 public:
  sc_port<fifo_if> fifo;
  sc_in_clk c_clock;
  SC_HAS_PROCESS(consumer);
  SC_MODULE(consumer){
    SC_THREAD(main);
    sensitive_pos << c_clock;
  }
  void main() {
    while(true) {
      wait();
      fifo->read(c);
      consume(c);
    } } };
```

FIGURE 7.3 – *The FIFO modules : buffer, producer and consumer [sys].*

```
int sc_main (int argc , char *argv[]) {
  fifo *fifo_inst;
  producer *prod_inst;
  consumer *cons_inst;
  sc_clock p_clock("ProducerClock", 10);
  sc_clock c_clock("ConsumerClock", 15);

  fifo_inst = new fifo("Fifo");

  prod_inst = new producer("Producer");
  prod_inst->fifo(*fifo_inst);
  prod_inst->p_clock(p_clock);

  cons_inst = new consumer("Consumer");
  cons_inst->fifo(*fifo_inst);
  cons_inst->c_clock(c_clock);

  sc_start(-1);
  return 0;
}
```

FIGURE 7.4 – *The FIFO main program [sys].*

The structure of the model is shown in Figure 7.2. The model contains a First-In-First-Out buffer and two modules cooperating through the buffer : a producer module which continuously puts data into the buffer and a consumer module which continuously retrieves data from the buffer. The two processes are triggered by two individual clocks : *p_clock* (to denote the producer clock) and *c_clock* (to denote the consumer clock). When a *p_clock* signal arrives, the producer starts producing and tries to write the product into the buffer. Similarly, the *c_clock* signal controls the consumer. The two clocks are independent, hence the producing and the consuming can be at different paces. It is certainly possible that the producer fills all the slots of the buffer and continues writing, or the consumer retrieves all the elements and still tries to consume. In this case, the producer (resp. consumer) must wait for the other to release (resp. fill) the buffer and this is done by the SystemC events mechanism.

Let us start by an informal analysis of the producer process as shown in Figure 7.5 : the producer is waiting for two main events : the *clock event* (*p_clock*) notified at each clock edge and the *r_event* (an event notified when the consumer reads an element from the buffer and we write *r_event* to designate a read event). The two wait statements define the two

waiting states of the automaton, and they divide the program into three pieces ($P1; P2; P3$ in Figure 7.5) according to the execution trace. Each piece of $P1$, $P2$ and $P3$ executes in an instant, and are seen actually as transitions between the waiting states. The objective is then

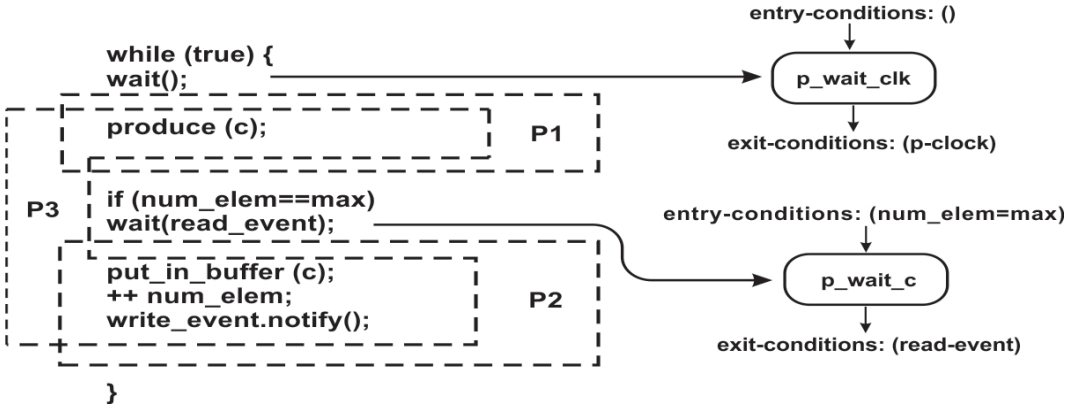


FIGURE 7.5 – *WSA generation of the producer.*

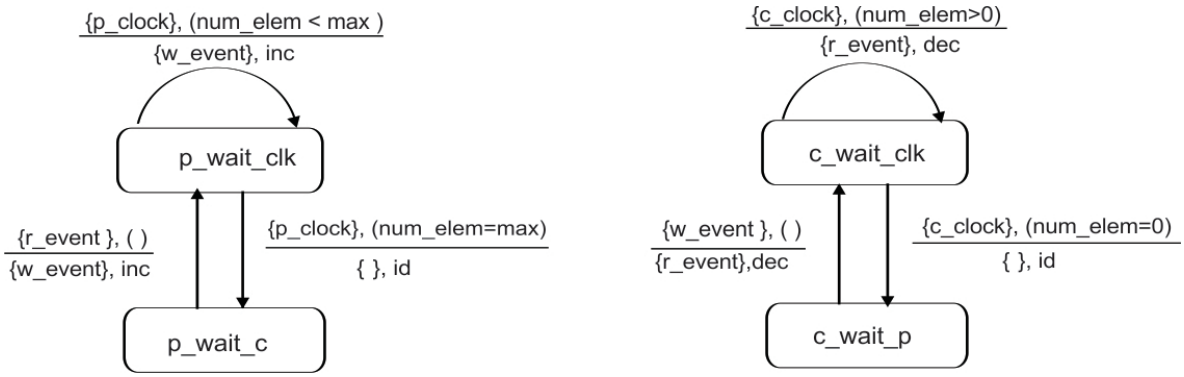


FIGURE 7.6 – *The WSA of the producer and the consumer.*

to represent formally how a process controls the transitions between the waiting-states : as shown in Figure 7.5, we calculate, for every waiting-state, the entry-conditions and the exit-conditions. The WSA of the producer, is composed of two waiting states s_1 and s_2. In s_1, the producer is waiting for the clock event to start execution, in s_2 it is waiting for an r_event : it is a special event triggered when the consumer read an element from the buffer. We define for each transition the entry and the exit conditions : set of conditions that respectively activates and is triggered when executing a transition. Intuitively, the events (p_clock, r_event) and the predicates over variables $p_1 = (num\_elem < max)$ and $p_2 = (num\_elem = max)$ act as guard conditions : e.g the transition from s_1 to s_2 is triggered if and only if the event p_clock

is present and the predicate p_1 holds. The event *w_event* and the function *inc* represent an effect : the transition will generate the event *w_event* and *inc* will be applied to the current instantiation of the variable *num_elem*.

7.3 Syntax

The SystemC waiting-state automata (WSA) is defined as a transition system A over a set $\mathcal{V}$ of variables. It is a tuple $A = (S; E; \mathcal{T})$, where S is a finite set of states, E a finite set of events and $\mathcal{T}$ a finite set of transitions where every transition is a 6-tuple $(s; e_{in}; p; e_{out}; f; s')$:

- s and s' are two states in S, representing respectively the initial state and the final state ;
- e_{in} and e_{out} are two sets of events : $e_{in} \subseteq E; e_{out} \subseteq E$;
- p is a predicate defined over variables in $\mathcal{V}$, i.e., $FV(p) \subseteq \mathcal{V}$, where *FV(p)* denotes the set of free variables in the predicate p ;
- f is an effect function over $\mathcal{V}$;

We write $s \xrightarrow[e_{out},f]{e_{in},p} s'$ for the transition $(s; e_{in}; p; e_{out}; f; s')$. The effect function set $\mathcal{F}(A)$ of the automaton $A(V)$ is the set of all effect functions in $A(\mathcal{V})$: $\mathcal{F}(A) = \{f | \exists t \in \mathcal{T} s.t.\ proj_6^5(t) = f\}$, where $proj_6^5$ denotes the fifth projection of a 6-tuple in the transition expression. We also write $proj_6^1$, $proj_6^2$, $proj_6^3$, $proj_6^4$, $proj_6^6$ to denote respectively the initial state s, the input event e_{in}, the predicate p, the output event e_{out} and the final state s'.

In the producer automaton, s and s' are respectively *p_wait_clk* and *p_wait_c*, the transition from s to s' is possible iff : the event *p_clock* is triggered and the variable $num\_elem = max$. As a consequence no events are triggered and the variable *num_elem* remains unchanged.

Definition 1 (The effect function) : The effect function is a first order logic formula that modifies the predicate p. We will define here the syntax of the function. But let's consider first the following notations :

- Variables (x, y, z, ...)

- Constants (a, b, c, ...)
- Predicates (p, q, r, ...)

Each fonction is a compostion of formulas defined over the predicates. Each predicate is composed of a combination of terms (variables and/or constants). The grammar of the function f is defined as below :

$$\begin{array}{lcl} \text{Term} & = & Constants|Variables \\ \text{Atom} & = & \text{Predicats} \\ \text{Formula} & = & Atom| \text{ Formula connector Formula} \\ \text{Connector} & = & +|-|*|\wedge|\vee \end{array}$$

In the FIFO example we have two operations on predicates : either to increment the predicate defined over the buffer elements or to decrement it. For ease of notation, we write the functions *inc* and *dec* to respectively increment and decrement the predicate *num_elem*. The effect functions *inc* and *dec* in the FIFO example are defined as follows :

- inc(num_elem) = num_elem + 1 ;
- dec(num_elem) = num_elem - 1.

We use the notation *id* to designate the identity function and *true* to designate the empty predicate.
We also use $\mathcal{T}(s)$ to denote the set of transitions from a given state s, i.e, $\mathcal{T}(s) = \{t|t \in \mathcal{T} \; and \; proj_6^1 = s\}$.

Definition 2 (Faithfulness of the automata) : In the SystemC WSA, the transition from a waiting state to another is only triggered by the events and the predicates determine which state the process will enter after being woken up, which means that transition from the same state must have the same set of incoming events e_{in}. Accordingly, we say that the SystemC waiting-state automaton $A = (S, E, \mathcal{T})$ is **faithful** to the initial process iff for every two transitions t and t' and for every state $s \in S$ we have :

$$\begin{cases} proj_6^1(t) = proj_6^1(t') \Rightarrow proj_6^2(t) = proj_6^2(t') \\ for \; every \; states \in \mathcal{S}, \bigvee_{t \in \mathcal{T}(s)} proj_6^3(t) \; always \; holds. \end{cases}$$

Definition 3 (minimal-step automata) : In fact, since the automata are derived by analyzing the waiting states of processes, as we have seen in the beginning of this section, a transition actually represents the execution within a minimal cycle, i.e., the execution of a process between two continuous *wait*. We call such an automaton a minimal-step waiting-state automata. We assume that every *minimal-step* automaton is total, i.e., every state has some successor. Otherwise, it means that the single process itself may cause a deadlock, which is not the case we study here.

7.4 Model Properties

The SystemC waiting-state automaton is an abstract composional model used to represent SystemC components. It is used to describe the functional behavior of SystemC processes at both the transaction level and the delta-cycle level. It is also used to verify non-functional behavior of SystemC constructs mainly the temporal behavior of critical real time systems.

- Abstraction of SystemC semantics : In order to make our verification methodology more efficient when dealing with SystemC designs, we give an abstract representation of SystemC designs that only includes the process related information (execution and activation events). To do so, we use an approach based on predicate abstraction [GS97].

- Compositionality : Components in a concurrent system interact with each other, and the correct functioning of different components is often mutually dependent. Therefore, achieving compositionality in the presence of concurrency is much more difficult than in sequential programming. Three different styles of verification methods with different degrees of compositionality are discussed in [WPdRP94]. They are named *global*, *modular* and *compositional* respectively : In a global method a concurrent system is modeled by a sequential one directly. A modular method typically consists of two steps : firstly, the processes are shown to be locally correct, and secondly, the local proofs are shown to be interference free with each other. In a compositional method, a component is developed in a way that the possible interference from its environment is already taken into account, so components are guaranteed to be interference free. Our model is supposed to verify the composionality property, we will verify this property in coming sections.

- Functional and non-functional properties : In formal verification techniques, it is necessary to verify functional properties of hardware/software systems to prove that the system is operating normally. But, non functional properties are also of fundamental relevance and imply a number of design decisions. The most important non-functional properties in this context are *synchronization*, *sharing*, *interaction*, *time properties* and *resources consumption*.

7.5 Conclusion

In this chapter, we presented the SystemC waiting-state automaton that we adopt to model SystemC designs at both the delta-level cycle and the transaction level. As we previously mentioned, the idea behind the model is first presented in [YZM07], where authors need to manually build the automata from SystemC programs. In this thesis, we extend the work of [YZM07] first to add more information about the time properties (Chapter 9) of parallel SystemC designs and second to propose an automatic approach to build, validate and verify a global framework modeled with the SystemC waiting-state automata (Part III and Part IV).

Later, we resume different algorithms to symbolically compose parallel automata and the algorithms to symbolically reduce the composed automaton. Then, we describe different extensions of the abstract model notably extensions using the parameters counter and time. We illustrate the use of each parameter on the Fifo example and we enumerate the properties that we can verify using each parameter.

CHAPTER 8

Symbolic Composition and Reduction of SystemC WSA

The global strategy of verifying SystemC models using the SystemC waiting-state automata is to define first a minimal-step automaton for every process, and then to compose them together so as to achieve a bigger automaton for the whole SystemC model which can be finally passed to the model-checking procedure. The symbolic composition follows the parallel composition of labeled Kripke structure as defined in [ECS04]. However, the symbolic composition applied in the SystemC WSA is followed by a reduction procedure, which enables more aggressive abstractions on the result model. Other typical abstractions include, for instance, replacing internal events of the composition with more abstract notions like counters, constraints on counters, time and constraints on time. With respect to the hiding of variables [KS05], such replacements keep functionality properties after the abstraction and introduce progressively QoS properties.

The SystemC waiting-state automaton are built separately for each SystemC process. The advantage of this approach is that the individual processes are much more smaller than the

overall program and verifying automata with less states is much more easier.

After building the automaton for each process separately, it is essential to apply parallel composition to minimal automata in order to build the abstract WSA for the whole system. The following formalizes steps for symbolic composition and reduction of the modular abstraction approach. We resume algorithms for the symbolic composition and reduction as first presented in [YZM07].

8.1 The Symbolic Composition of the SystemC WSA

The symbolic composition step consists in building a large automaton for the whole program, where minimal SystemC waiting-state automata for the processes are composed together in a bottom-up approach. Below is defined the algorithm for the symbolic composition where all the combinations of process states are considered. The symbolic composition is followed by a reduction procedure, which enables more aggressive abstractions on the result model.

Algorithm Let's consider two SystemC waiting-state automata $A = (S, E, \mathcal{T})$ and $A' = (S', E', \mathcal{T}')$ over the same set $\mathcal{V}$ of variables, the resulting automaton is a tuple $(S \times S', E \cup E', \mathcal{T}")$ defined as below :

- ❑ $(s_1, s'_1) \xrightarrow[e_{out}, f]{e_{in}, p} (s_2, s'_1) \in \mathcal{T}"$ for every state $s_1 \xrightarrow[e_{out}, f]{e_{in}, p} s_2 \in \mathcal{T}$ and $s'_1 \xrightarrow[e'_{out}, f']{e'_{in}, p'} s'_2 \in \mathcal{T}'$, either $e'_{in} \not\subseteq e_{in}$ or $p \not\Rightarrow p'$.
- ❑ $(s_1, s'_1) \xrightarrow[e'_{out}, f']{e'_{in}, p'} (s_1, s'_2) \in \mathcal{T}"$ for every state $s_1 \xrightarrow[e_{out}, f]{e_{in}, p} s_2 \in \mathcal{T}$ and $s'_1 \xrightarrow[e'_{out}, f']{e'_{in}, p'} s'_2 \in \mathcal{T}'$, either $e_{in} \not\subseteq e'_{in}$ or $p' \not\Rightarrow p$.
- ❑ $(s_1, s'_1) \xrightarrow[e_{out} \cup e'_{out}, f \circ f']{e_{in} \cup e'_{in}, p \wedge p'} (s_2, s'_2) \in \mathcal{T}"$ for every state $s_1 \xrightarrow[e_{out}, f]{e_{in}, p} s_2 \in \mathcal{T}$ and $s'_1 \xrightarrow[e'_{out}, f']{e'_{in}, p'} s'_2 \in \mathcal{T}'$.

In other words, components must synchronize on shared actions and proceed independently on local actions. Nevertheless, SystemC processes running in parallel may cause causality waiting cycles and it is also represented in the composition of SystemC waiting-state automata. The verification requires in the first place to detect the *unsafe states* that contain mutually waiting processes, for further analysis based on the automata. Besides, the symbolic composition of the minimal-step automata is guaranteed to be recursive within each SystemC module.

Definition 1 (unsafe states) : We say that a state (s_1, s'_1) in the composed automaton is a potential *unsafe* state if $e_{in}(s_1) \cap e_{out}(s'_1) \neq \emptyset$ and $e_{in}(s'_1) \cap e_{out}(s_1) \neq \emptyset$. For instance, in the FiFo example, the composition of the producer automaton and the consumer automaton as shown in Figure 8.1 gives rise to a potential unsafe state $(p\_wait\_c; c\_wait\_p)$. Indeed, it is an *unsafe state* where the two processes are waiting for each other.

Definition 2 (non-deterministic transitions) : During symbolic composition, it is possible to replace the effect function $f \circ f'$ of the new transition by $f' \circ f$, but the composed automaton might not be equivalent since $f \circ f'$ and $f' \circ f$ are not always equal. This is the case where the composition will result in potential *non-deterministic* behavior. The transition defined with an effect function $f \circ f'$ such that $f \circ f' \neq f' \circ f$ is called a non-deterministic transition.
Let's take the following example where we consider two functions f and f' defined over the same variable x such as : f=x+1 and f'=x-1

$$f \circ f' = f(f') = (x-1)+1 = x \; and \; f' \circ f = f(f') = (x+1)-1 = x.$$

In Figure 8.1, the transition $(p\_wait\_clk, c\_wait\_p) \xrightarrow[\{r\_event, w\_event\}, num\_elem{+}{+} \& num\_elem{-}{-}]{\{p\_clk, w\_event\}, (num\_elem < max)}$ (p_wait_clk, c_wait_clk) is a deterministic transition since $(num\_elem{+}{+}) \circ (num\_elem{-}{-}) = (num\_elem{-}{-}) \circ (num\_elem{+}{+}$.

The symbolic composition of SystemC automata can be used to check the *determinism* of a SystemC model : First, the corresponding minimal-step automaton for every process is defined. Next, all automaton are composed together ; if the composed automaton does not contain any non-deterministic transition, one can assert that the model is *deterministic*. Detecting non-deterministic transitions can be done without doing the composition. Because the above definition includes all composes of effect functions of the two component automata. One can simply check whether $f \circ f' = f' \circ f$, where $f \in \mathcal{F}(A); f' \in \mathcal{F}(A')$ (A, A' are two component automata). However, such a detection might be too strict in the sense that some non-deterministic transitions may never be triggered and does not change the deterministic behavior of the model. This is often because the guard condition of these *impossible transitions* will never be true, e.g., p and p' in the above definition are not true. Actually, such transitions can be removed after the composition as a refinement, and clearly, checking the non-existence of non-deterministic transitions based on the refined composition will increase the precision

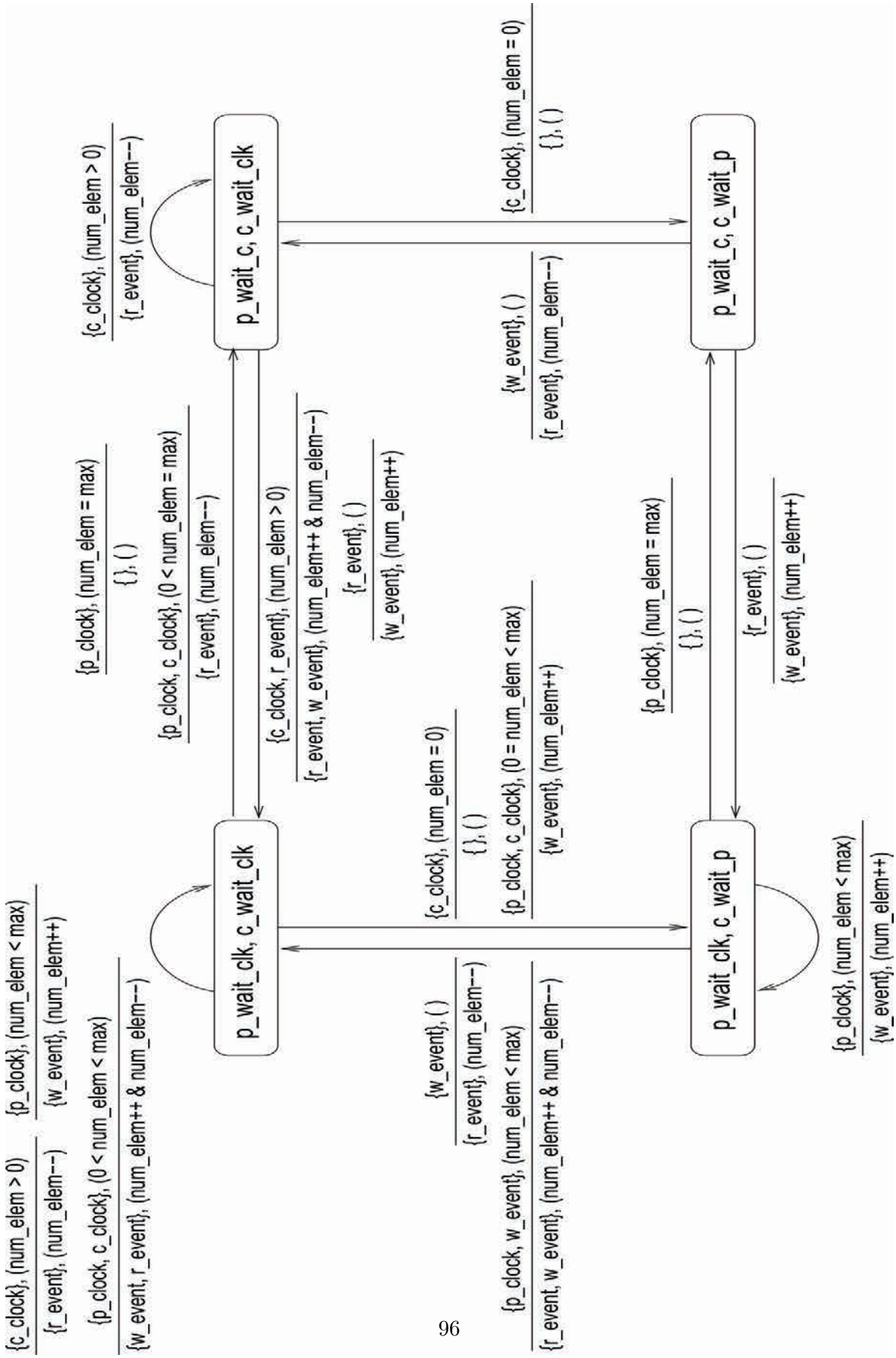


FIGURE 8.1 – *The composed SystemC waiting-state automaton for the FIFO example*

of the detection of the non-determinism of SystemC models.

8.2 The Symbolic Reduction of the SystemC WSA

During symbolic composition, all possible transitions between symbolic states are generated. Transitions include *safe* transitions, *impossible* transitions, *redundant* transitions and *reducible* transitions. Let's define first each category of transitions.

- *The safe transitions* : They represent the set of possible transitions generated during symbolic execution. Those transitions are usually triggered in both the minimal-step automata and the composed automaton.
- *The impossible transitions* : They represent the set of transitions that can never be triggered in the composed automata. They are impossible either because their corresponding entry-conditions can never hold, or because they correspond to unsafe states as previously explained.
- *The redundant transitions* :They represent the set of transitions that have the same entry and exit conditions. In this case, it is better to keep only one of these redundant transitions.
- *The reducible transitions* : They represent a sequence of consecutive transitions that are inter-independent ; i.e, the exit-conditions of one transition represent the entry-conditions for the consecutive transition. In this case, all transitions are merged together and transformed into only one transition.

Symbolic reduction is a later stage that consists in keeping track of only safe transitions. It is a major step to build the final automaton. Thus, it reduces all impossible transitions, replaces redundant transitions and manages the set of reducible transitions. It considers the environment influence on the system execution, i.e, set of behaviors that may not happen in the composed automata. Besides, reduction consists in the concatenation of transitions, i.e, the affect of a certain transition will immediately trigger another transition, we may replace both transitions by a new one. For example we consider the following two transitions :

$$s_1 \xrightarrow[e_{out},f]{e_{in},p} s_2 \xrightarrow[e'_{out},f']{e'_{in},p'} s_3$$

Where $e'_{in} \subseteq e_{out}$ and $p \Rightarrow f(p)$. In this case, we may replace the two transitions (called *reducible* transitions) with a new transition $s_1 \xrightarrow[e'_{out},f' \circ f]{e_{in} \cup e_{in},p} s_2$ (which is called the *contractum* of them).

In the FIFO example, we can find examples of such reducible transitions, e.g., in the composed automaton of the two minimal-step automata (see Figure 8.1), the transition : $(p\_wait\_clk, c\_wait\_clk) \xrightarrow[\{w\_event\},inc]{\{p\_clk,c\_clk\},(0=num\_elem<max)} (p\_wait\_clk, c\_wait\_p)$ will immediately trigger the transition $(p\_wait\_clk, c\_wait\_p) \xrightarrow[\{r\_event\},dec]{\{w\_event\},()} (p\_wait\_clk, c\_wait\_clk)$ so we can replace both of them by a new transition $(p\_wait\_clk, c\_wait\_clk) \xrightarrow[\{r\_event\},incoded]{\{p\_clk,c\_clk\},(0=num\_elem<max)}$ (p_wait_clk, c_wait_clk).

Besides, in both Figure 8.2 and Figure 8.1, we have other examples of reducible transitions such as : $(p\_wait\_clk, c\_wait\_p) \xrightarrow[\{w\_event\},inc]{\{p\_clk\},(num\_elem<max)} (p\_wait\_clk, c\_wait\_p)$ that immediately triggers the transition $(p\_wait\_clk, c\_wait\_p) \xrightarrow[\{r\_event\},dec]{\{w\_event\},()}$ (p_wait_clk, c_wait_clk). Both of them, can be replaced by the following transition : $(p\_wait\_clk, c\_wait\_p) \xrightarrow[\{r\_event\},dec \circ inc]{\{p\_clk\},(num\_elem<max)} (p\_wait\_clk, c\_wait\_clk)$. Besides, the transition $(p\_wait\_c, c\_wait\_clk) \xrightarrow[\{r\_event\},dec]{\{c\_clk\},(0<num\_elem)} (p\_wait\_c, c\_wait\_clk)$ can immediately trigger the transition $(p\_wait\_c, c\_wait\_clk) \xrightarrow[\{w\_event\},inc]{\{r\_event\},()} (p\_wait\_clk, c\_wait\_clk)$ and both of them can be concatenated into the transition $(p\_wait\_c, c\_wait\_clk) \xrightarrow[\{w\_event\},dec \circ inc]{\{c\_clk\},(0<num\_elem)}$ (p_wait_clk, c_wait_clk).

In general, a minimal-step waiting-state Automaton does not contain any reducible transitions. During the verification strategy, reductions are usually required when composing all the minimal-step automata together. As its is intend to define a model that represents the behavior at the level of delta-cycles and hides all interactions within a delta-cycle, the reduction algorithm should be consistent with the SystemC scheduler and it must shows all possible interactions between the two processes within a single delta-cycle.

Algorithm Given a SystemC minimal-step waiting-state automaton $A(V) = (S; E; \mathcal{T})$, where $\mathcal{T}$ has reducible transitions, let $\mathcal{T}_0 := \mathcal{T}$, $\mathcal{T}_{remove} := \{\}$ and $\mathcal{T}_{new} := \{\}$. The following steps define the algorithm to build the reduced automaton from the automaton generated during symbolic execution :

1. for every reducible pair (t_1, t_2) and its contractum t_3, where $t_1, t_2 \in \mathcal{T}_0$, let $\mathcal{T}_{remove} :=$

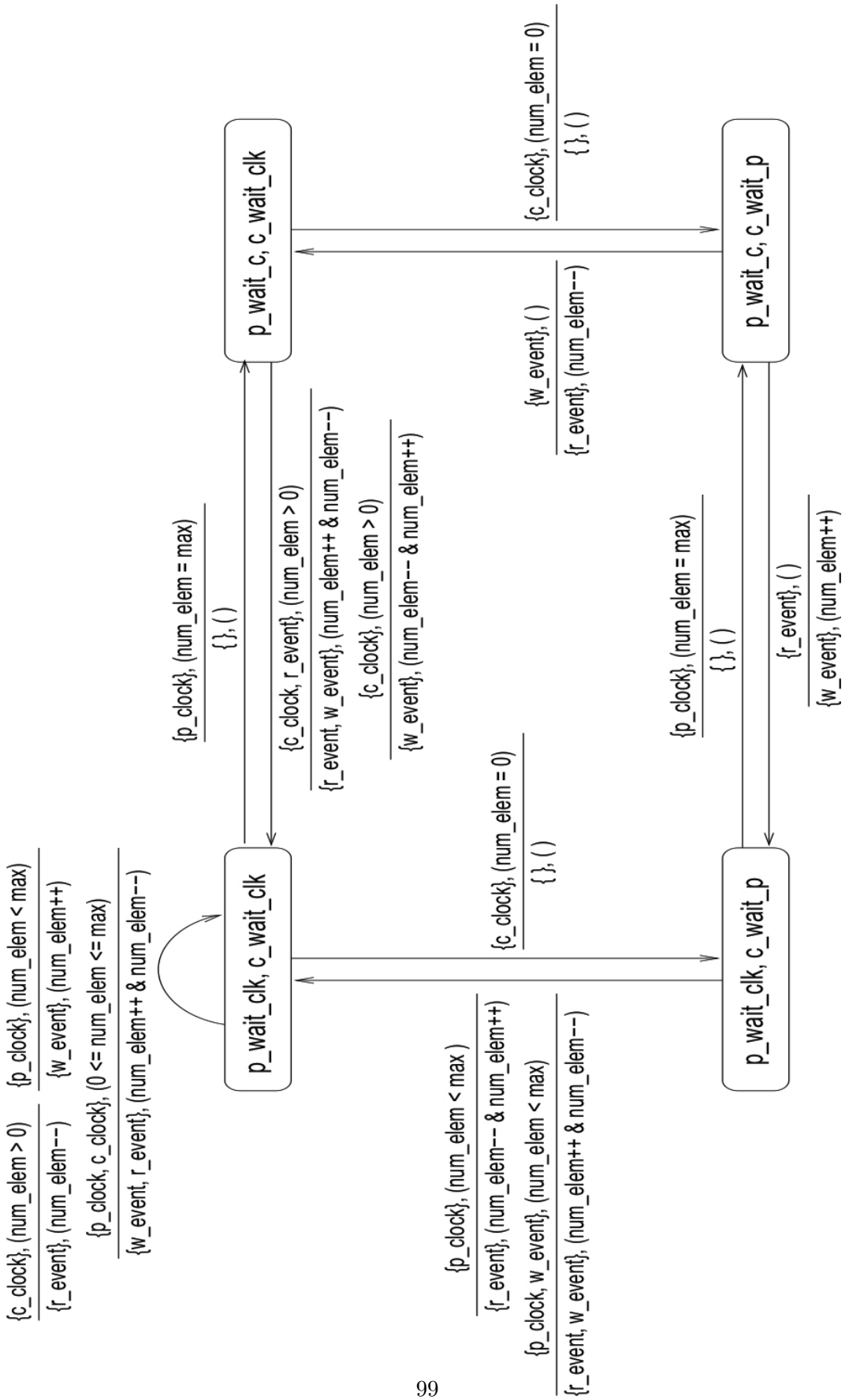


FIGURE 8.2 – *The reduced SystemC waiting-state automaton for the FIFO.*

$\mathcal{T}_{remove} \cup t_1, t_2$ and $\mathcal{T}_{new} := \mathcal{T}_{new} \cup t_3$;

2. repeat the above step until all reducible pairs in T_0 has been manipulated;
3. let $\mathcal{T}_0 := (\mathcal{T}_0/\mathcal{T}_{remove}) \cup \mathcal{T}_{new}$, $\mathcal{T}_{remove} := \{\}$ and $\mathcal{T}_{new} := \{\}$;
4. if there are still reducible pairs in $\mathcal{T}_0$, go to the step 1 and repeat the above procedure; otherwise, let $\mathcal{T}' := \mathcal{T}_0$.

The reduced automaton is $(S, E, \mathcal{T}')$.

If we consider n processes $(P_1, \ldots, P_n)$ and $(W_1, \ldots, W_n)$ their corresponding SystemC waiting-state automata respectively. Let W be the reduced automaton of $(W_1 \times, \ldots, \times W_n)$ using the algorithm for symbolic reduction, then every transition in W represents a whole execution of the SystemC model within a delta cycle. Otherwise, for every transition such that : $s_1 \xrightarrow[e_{out},f]{e_{in},p} s_2$, if at the beginning of a delta cycle the model is in a state s_1 and the predicate p holds, and if all events in e_{in} are provided by the environment, then at the end of a delta cycle, the model will be in the state s_2.

A formal modeling of SystemC processes is done up to the level of delta cycles, using the SystemC waiting-state automata, together with the composition and the reduction algorithms. Note that at this stage, events of the final automaton can be devided into two sets : the set of *environment* events E_e and the set of *internal* events E_i. The environment events are events generated by the SystemC engine, which are typically *time events* such as clock events. Using the latter classification of events, the SystemC waiting-state automaton can be reduced again by removing those transitions depending on non environment events, i.e., transitions where $e_{in} \notin E_e$. For instance, the reduced automaton in Figure 8.2 can be again reduced to the one in Figure 8.3. We take for example, the transition : $t_1 = (p\_wait\_c, c\_wait\_p) \xrightarrow[\{r\_event\},dec]{\{w\_event\},()} (p\_wait\_c, c\_wait\_clk)$, that depends on w_event which presents the entry condition for this transition. Since, the generation of the event w_event depends on the previous transitions that immediately trigger t_1, i.e, w_event must figure in the set of *exit conditions* of these transitions. We conclude that t_1 must be reduced from the automaton in Figure 8.2, the same applies to transitions : $(p\_wait\_c, c\_wait\_p) \xrightarrow[\{w\_event\},inc]{\{r\_event\},()} (p\_wait\_clk, c\_wait\_p)$, $(p\_wait\_clk, c\_wait\_p) \xrightarrow[\{r\_event,w\_event\},dec \circ inc]{\{p\_clk,w\_event\},(num\_elem<max)} (p\_wait\_clk, c\_wait\_clk)$ and finally the transition $(p\_wait\_c, c\_wait\_clk) \xrightarrow[\{r\_event,w\_event\},inc \circ dec]{\{c\_clk,r\_event\},(num\_elem>0)} (p\_wait\_clk, c\_wait\_clk)$.

Interestingly, we can conclude easily from the new automaton that the state (p_wait_c, c_wait_p) is an unsafe state since there is no transition getting out of it. Indeed, this state is exactly the deadlock state where the producer and the consumer are waiting for each other.

8.3 Conclusion

The main idea behind the use of a bottom-up approach for system modeling is to build a global automaton by composing a set of parallel automata : this was the main goal of this chapter. Indeed, we resume the definition of the symbolic composition and reduction of the SystemC waiting-state automata as first presented in [YZM07]. The goal behind the symbolic composition is first to compose different states of different components. Then, we generate different transitions between the composed states. Finally, we determine the set of unsafe states and we reduce them, we also eliminate non-deterministic transitions using the definitions we mentioned before.

During the symbolic reduction step, we eliminate the set of unfaisable transitions and we reduce the set of the reducible and the redundant transitions. We also consider the effect of the environment events. We define different steps for the symbolic composition and reduction and we illustrate that on the Fifo example.

In the next chapter, we define possible extensions of the SystemC WSA using parameters counter and time. We also define how to enhance the previous algorithms for the symbolic composition and reduction using the extended automata.

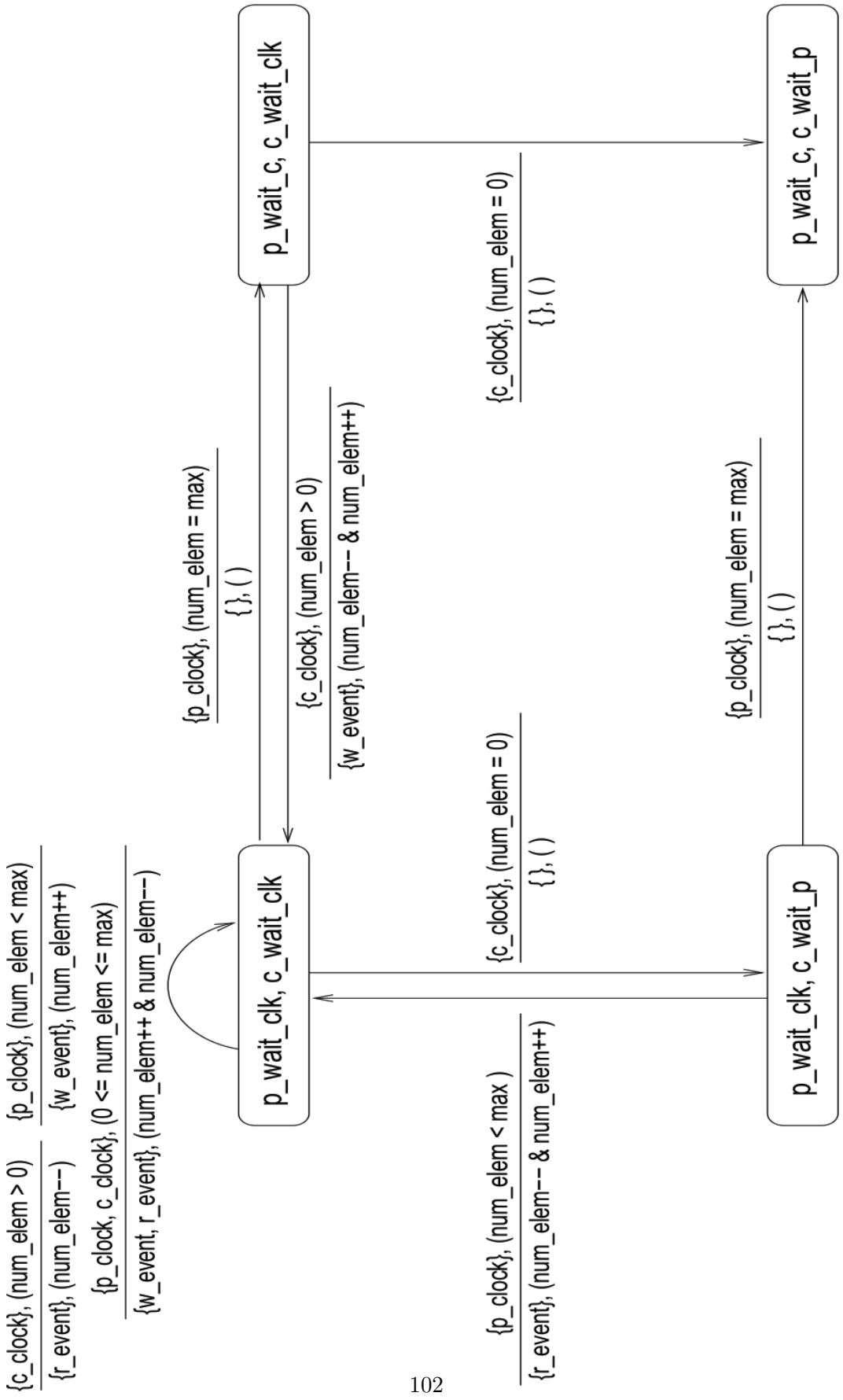


FIGURE 8.3 – *The environment-sensitive SystemC waiting-state automaton for the FIFO.*

CHAPTER 9

Extending the SystemC Waiting State Automata with Counters and with Time

Parametric automata are used in various synthesis problems. They are also used to model programs, whose behavior depend on inputs values from the environment [RAV93]. Parameters are also used to model resources (e.g., time, memory) consumed by transitions.

In this chapter, we start from the previous extension of the SystemC waiting-state automata as presented in [YZM07], the automata are extended with counters. Counter automata are basically used to model concurrent distributed systems and to verify properties like the *reachability* problem, *liveness* and *determinism*. Authors in [YZM07] use counters to verify further properties on the SystemC waiting-state automata ; one of these properties is to infer the relations between the entry and the exit conditions. In this thesis, we resume the same definition of counter automata as in [YZM07]. But we do more, first we develop further the use of counters on the automata and we specify on some examples the use of the parameter. Second, we extend the model with further parameters ; i.e, we annotate the model with temporal information, which was not done in the previous work. We denote respectively by (δ), *(t)* and *(d)* the *counter*, the starting time and the duration of the associated transition. Each parameter is defined on a transition : (δ) represents the number of time the transition was triggered, (t) is the time when the transition starts and (d) is the duration of the transition once triggered. We present later the use of each parameter separately and then the use of all the parameters together.

9.1 Extending the SystemC WSA with Counters

Counter automata are widely used formalisms to model concurrent distributed systems. Basically, a counter automaton is a finite-state automaton annotated with counters that hold positive integer values. We apply arithmetic operations on counters. Counter automata are naturally *infinite-state* systems since the counters are unbounded numbers. They are used to model some desired properties on systems. Among these properties, we may mention *safety* properties : these properties may often be expressed by *reachability* properties on the model. Reachability properties are algorithmically checkable for *finite-state* systems, however this situation is more complex for *infinite-state* systems.

In the SystemC waiting-state automata, each transition is annotated with a counter, which is a global variable that is incremented by 1 each time the transition is triggered. This

parameter is used first to infer the relations between system transitions and second to verify properties about behavioral relations between processes and about processes themselves :

- If we take each process independently : δ counts how many times the predicate p is true, how many times the event e_{out} was triggered and changes the affect of the function *f(p)*. All these information will be used later to decide for example which transition is safe and accordingly decide whether a state is reachable or not.

- If we take the composed automaton (for the whole system) : δ is used to reduce transitions that don't satisfy conditions on counters. We can also use counters to detect deadlocks on the composed automata, decide about safe states, reduce impossible and redundant transitions.

We resume the example of the FIFO, where we have two main processes : the producer that is writing an element to the buffer at each *p_clock* and the consumer which is reading an element from the buffer at each *c_clock*. We take each automaton separately and we extend it with counters : a counter for each transition. Figure 9.1 shows the automata of the producer and the consumer extended with counters. If we take for example the parameter δ_w^1 : it counts

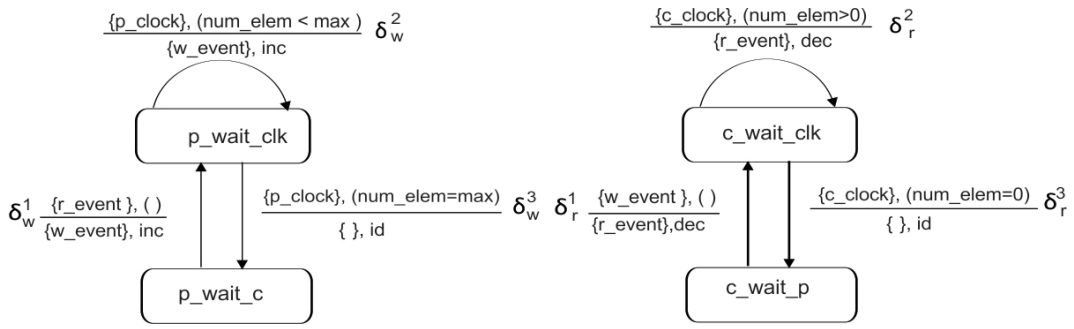


FIGURE 9.1 – *The automata for the consumer and the producer extended with counters.*

the number of times the producer entered the wait state *p_wait_clk*, i.e, how many times it was activated. Besides, a *read* event can only occur if a *write* occurs before it. In other words, if we want to verify the relation between the notified events *r_event*, *w_event* and the variable *num_elem* which determine the number of elements in the buffer, we have to verify the relation below : the number of elements read from the buffer must be less or equal to the number of

elements written to the buffer. Formally, it should always hold that :

$$\begin{cases} \delta_w^1 + \delta_w^2 \geq \delta_r^1 + \delta_r^2 \\ (\delta_w^1 + \delta_w^2) - (\delta_r^1 + \delta_r^2) \leq num\_elem \end{cases} \tag{9.1}$$

We may add condition on counters to verify entry and exit conditions, e.g, the condition to exit the state *p_wait_clk* is the presence of the signal *p_clk* and the condition $\delta_w^1 = \delta_w^3 - 1$.

Such relations on counters can be inferred during the construction stage of the WSA using for instance techniques based on abstract interpretation [CC77, Bal02, Moy05] like in [Ven98, Ven97].

9.1.1 Syntax

An extended SystemC waiting-state automaton over a set $\mathcal{V}$ of variables, is a quadruple $A = (S, E, \mathcal{T}, \mathcal{C})$, where S is a finite set of states, E is a finite set of events, $\mathcal{C}$ is a finite set of counters and $\mathcal{T}$ is a finite set of transitions where every transition is a 7-tuples $(s, e_{in}, p, e_{out}, f, s', \delta)$:

- s and s' are two states in S, representing respectively the initial state and the final state of the transition,
- e_{in} and e_{out} are two sets of events : $e_{in} \subseteq E, e_{out} \subseteq E$,
- p is a predicate defined over variables in $\mathcal{V}$ and counters $\mathcal{C}$, i.e, $FV(p) \subseteq \mathcal{V} \bigcup \mathcal{C}$,
- f is an effect function over $\mathcal{V}$,
- $\delta \in \mathcal{C}$ is the counter associated to the transition that increments each time it is executed.

The strategy of modeling SystemC designs using the extended SystemC waiting-state automata is similar as before : it requires first to build a minimal-step automaton for each process, annotate the transitions with counters. The counters are used to specify conditions about the number of time a transition can be activated during the execution of the model. Once conditions on counters are well defined (one can resort to abstract techniques to approve those conditions), we proceed to the symbolic composition of the minimal-step automata generated for each process. A later stage is to reduce the composed automaton using the symbolic reduction algorithm. The automaton can then be passed to a model-checking procedure.

9.1.2 The Symbolic Composition

The symbolic composition of two automata A and A' over the same set of variables $\mathcal{V}$ works similarly as in the case of standard waiting-state automata. What makes the composition more complicated is that parameters of the automata A and A' do not correspond to the parameters of the composed automaton $A \times A'$. A transition τ of the automata A may be composed with many transitions $\{\tau'_1, \ldots, \tau'_n\}$ of the automaton A', and the values of counters of the transition τ in A should be represented in the values of counters for $(\tau \times \tau'_0), \ldots, (\tau \times \tau'_n)$.

If $\delta_\tau^{\tau'_k}$ denote the counter associated to the transition $(\tau \times \tau'_k)$ in the composed automata, then $\delta_\tau = \sum_k \delta_\tau^{\tau'_k}$.

As the transition predicates (i.e., guard conditions) in the extended automata are defined over counters and system variables, the composition should ensure that these predicates of component automata are properly translated in the composed automaton. In particular, we must replace all the occurrence of a transition counter from the component automata, with the value δ_τ recently presented.

Algorithm Given two extended SystemC waiting-state automata $A = (S, E, \mathcal{T}, \mathcal{C})$ and $A = (S', E', \mathcal{T}', \mathcal{C}')$, over the same set $\mathcal{V}$ of variables. The combination of the two SystemC waiting-state automata is still a SystemC waiting-state automaton $(S \times S', E \bigcup E', \mathcal{T}'', \mathcal{C}'')$ written as $A \times A'$ where $\mathcal{T}''$ is the smallest set of transitions and $\mathcal{C}''$ is the associated set of counters.$\Pi(s, e_{in}, p, e_{out}, f, s', \delta)$ is the set of counters of $\mathcal{C}''$ associated to a transition in $\mathcal{T} \times \mathcal{T}'$. M_c a morphism that maps counters in $\mathcal{C} \times \mathcal{C}'$ to $\mathcal{C}''$, such that :

- $\Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta) := \{\delta^*\} \bigcup \Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta)$ and $(s_1, s'_1) \xrightarrow[e_{out}, f]{e_{in}, M_c(p), \delta^*} (s_2, s'_1) \in \mathcal{T}''$ for every transition $s_1 \xrightarrow[e_{out}, f]{e_{in}, p, \delta} s_2 \in \mathcal{T}$ and for every state $s'_1 \in S'$ such that for every transition $s'_1 \xrightarrow[e'_{out}, f', s'_2]{e'_{in}, p', \delta'} s'_2 \in \mathcal{T}'$, either $e'_{in} \nsubseteq e_{in}$ or $p \not\Rightarrow p'$,
- $\Pi(s'_1, e'_{in}, p', e'_{out}, f', s'_2, \delta') := \{\delta^*\} \bigcup \Pi(s'_1, e'_{in}, p', e'_{out}, f', s'_2, \delta')$ and $(s_1, s'_1) \xrightarrow[(e'_{out}, f')]{e'_{in}, M_c(p'), \delta^*} (s_1, s'_2) \in \mathcal{T}''$ for every transition $s'_1 \xrightarrow[e'_{out}, f']{e'_{in}, p', \delta'} s'_2 \in \mathcal{T}'$ and for every state $s_1 \in S$ such that for every transition $s_1 \xrightarrow[e_{out}, f]{e_{in}, p, \delta} s_2 \in \mathcal{T}$, either $e_{in} \nsubseteq e'_{in}$ or $p' \not\Rightarrow p$,
- $\Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta) := \{\delta^*\} \bigcup \Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta)$ and
  $\Pi(s'_1, e'_{in}, p', e'_{out}, f', s'_2, \delta') := \{\delta^*\} \bigcup \Pi(s'_1, e'_{in}, p', e'_{out}, f', s'_2, \delta')$ and

$(s_1, s_1') \xrightarrow[e_{out} \bigcup e'_{out}, f \circ f']{e_{in} \bigcup e'_{in}, M_c(p \wedge p'), \delta^*} (s_2, s_2') \in \mathcal{T}''$, for every pair of transitions $s_1 \xrightarrow[e_{out}, f]{e_{in}, p, \delta} s_2 \in \mathcal{T}$ and $s_1^{'} \xrightarrow[e'_{out}, f']{e'_{in}, p', \delta'} s_2' \in \mathcal{T}'$.

- ❑ According to the transition $(s_1, e_{in}, p, e_{out}, f, s_2, \delta_{s_1}^{s_2})$, the morphism M_c maps the counter δ to the *sum* of transition counters defined in $\Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta)$

$$M(\delta) \rightarrow \sum_{\delta^* \in \Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta_{s_1}^{s_2})} \delta^*.$$

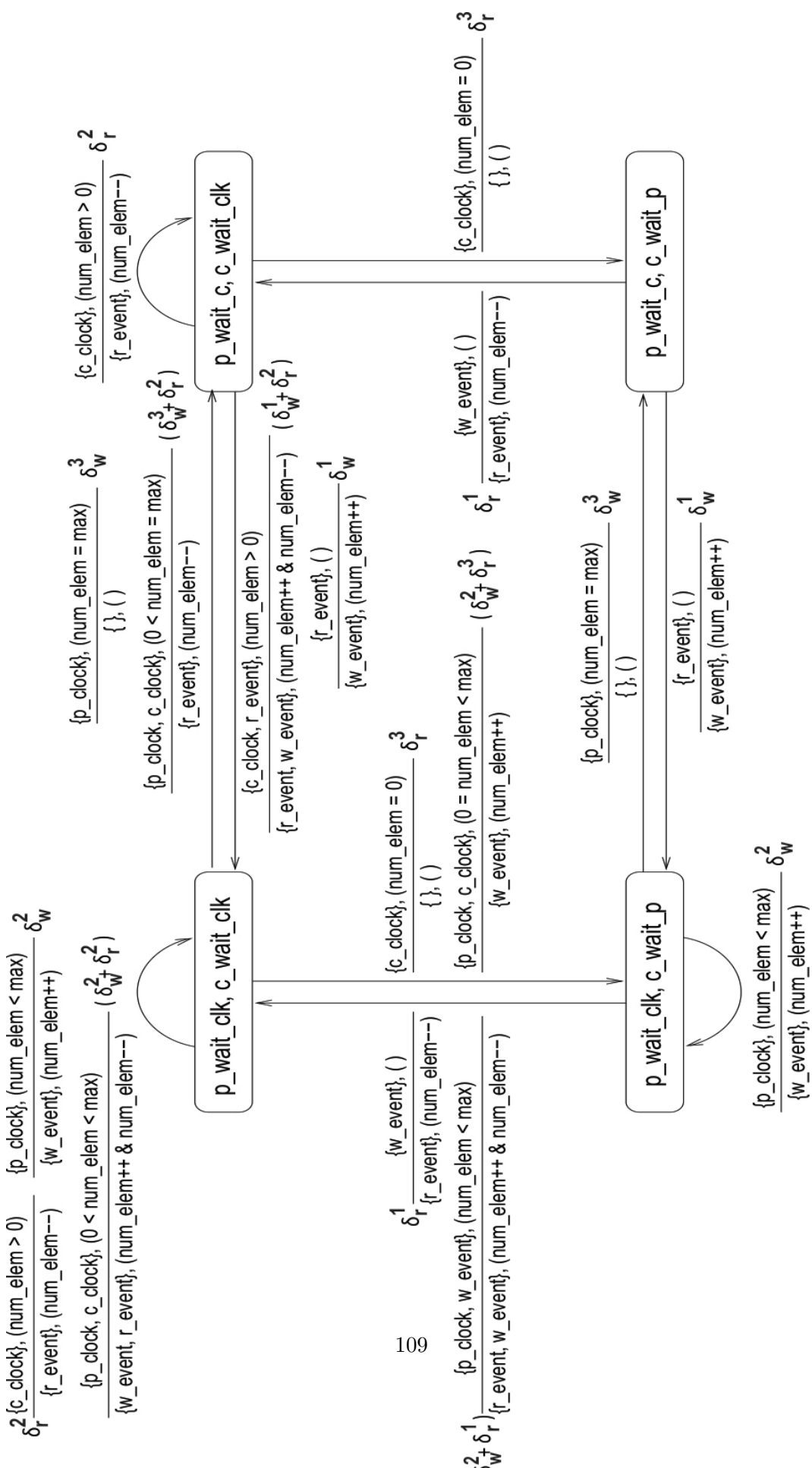
Definition 1 A morphism M is an abstraction structure that maps between two mathematical structures (called objects). Much of the terminology of morphisms comes from concrete categories, where objects are simply sets with some additional structure, and morphisms are functions preserving this structure.

A morphism is often thought of as an arrow linking an object called the **domain** to another object called the **codomain**. In set theory, morphisms are functions ; in topology, morphisms are continuous functions ; in universal algebra, they are called homomorphisms ; in group theory, we call them group homomorphisms.

Definition 2 There are two operations defined on every morphism, the domain (or source) and the codomain (or target). If a morphism f has domain X and codomain Y, we write $f : X \rightarrow Y$. Thus a morphism is an arrow from its domain to its codomain. The set of all morphisms from X to Y is denoted $homC(X, Y)$ or simply $hom(X, Y)$ and called the hom-set between X and Y.
For every three objects X, Y, and Z, there exists a binary operation $hom(X, Y) \times hom(Y, Z) \rightarrow hom(X, Z)$ called composition. The composition of morphisms is often represented by a commutative diagram. For example, Morphisms satisfy two axioms :

- ❖ *Identity* : for every object X, there exists a morphism $idX : X \rightarrow X$ called the identity morphism on X, such that for every morphism we have $idB \circ f = f = f \circ idA$.
- ❖ *Associativity* : $h \circ (g \circ f) = (h \circ g) \circ f$ whenever the operations are defined.

Figure 9.2 shows the composed automaton for the FIFO example annotated with counters. The automaton is similar to the automaton in Figure 8.1 to which are added relations between


p_wait_clk, c_wait_clk
p_wait_c, c_wait_clk
p_wait_clk, c_wait_p
p_wait_c, c_wait_p
{c_clock}, (num_elem > 0) / {r_event}, (num_elem--) δ2r
{p_clock}, (num_elem < max) / {w_event}, (num_elem++) δ2w
{p_clock, c_clock}, (0 < num_elem < max) / {w_event, r_event}, (num_elem++ & num_elem--) (δ2w+ δ2r)
{p_clock}, (num_elem = max) / { }, ( ) δ3w
{p_clock, c_clock}, (0 < num_elem = max) / {r_event}, (num_elem--) (δ3w+ δ2r)
{c_clock}, (num_elem > 0) / {r_event, w_event}, (num_elem++ & num_elem--) (δ1w+ δ2r)
{r_event}, ( ) / {w_event}, (num_elem++) δ1w
{c_clock}, (num_elem > 0) / {r_event}, (num_elem--) δ2r
{c_clock}, (num_elem = 0) / { }, ( ) δ3r
{p_clock, c_clock}, (0 = num_elem < max) / {w_event}, (num_elem++) (δ2w+ δ3r)
δ1r {w_event}, ( ) / {r_event}, (num_elem--)
{p_clock}, (num_elem = max) / { }, ( ) δ3w
{r_event}, ( ) / {w_event}, (num_elem++) δ1w
{p_clock}, (num_elem < max) / {w_event}, (num_elem++) δ2w
δ2w+ δ1r) {p_clock, w_event}, (num_elem < max) / {r_event, w_event}, (num_elem++ & num_elem--)

counters of minimal automata.

9.1.3 The Symbolic Reduction

The process of symbolic reduction of WSA extended with counters is similar to that of normal WSA, since reduction eliminates impossible transitions. In fact, if the effect of a certain transition will immediately trigger another transition, we may replace both transitions by a new one ; we call that the concatenation of two transitions. But, in this case we have counters or conditions over counters that impact the reduction process.
If we resume the FIFO example and take the example above of reduced transitions presented in the previous section such that : $t_1 = (p\_wait\_clk, c\_wait\_clk) \xrightarrow[\{w\_event\},inc]{\{p\_clk,c\_clk\},(0=num\_elem<max)} (p\_wait\_clk, c\_wait\_p)$ that will immediately trigger the transition $t_2 = (p\_wait\_clk, c\_wait\_p) \xrightarrow[\{r\_event\},dec]{\{w\_event\},()} (p\_wait\_clk, c\_wait\_clk)$ so we can replace both of them by a new transition $t_3 = (p\_wait\_clk, c\_wait\_clk) \xrightarrow[\{r\_event\},incoded]{\{p\_clk,c\_clk\},(0=num\_elem<max)} (p\_wait\_clk, c\_wait\_clk)$ We suppose that δ_1, δ_2 and δ_3 are respectively the counters associated to t_1, t_2 and t_3. In the reduced automata, δ_3 must verify : $\delta_3 = \delta_1 + \delta_2$. Besides, in the reduced automaton mentioned in Figure 8.3 we add another condition over counters, i.e, the sum of counters representing a *w_event* must be superior or equal to the sum of counters representing a *r_event* in the reduced automaton. Moreover, the sum of counters presenting a *w_event* and those presenting a *r_event* must be less or equal to the number of elements in the buffer (similar to equation (1)).

Algorithm Given a SystemC minimal-step waiting-state automaton $A = (S, E, \mathcal{T}, \mathcal{C})$, where $\mathcal{T}$ has reducible transitions, let $\mathcal{T}_0 := \mathcal{T}$, $\mathcal{C}_0 := \mathcal{C}$, let $\mathcal{T}_{remove}$, $\mathcal{T}_{new}$, $\mathcal{C}_{remove}$, $\mathcal{C}_{new} := \{\}$. The following steps define an algorithm of removing the reducible transitions :

1. For every reducible pair (t_1, t_2) and its contractum t_3, where $t_1, t_2 \in \mathcal{T}_0$, let : $\mathcal{T}_{remove} := \mathcal{T}_{remove} \bigcup \{t_1, t_2\}$ and $\mathcal{T}_{new} := \mathcal{T}_{new} \bigcup \{t_3\}$, $\mathcal{C}_{remove} = \mathcal{C}_{remove} \bigcup \{$ the counters associated to t_1 and $t_2\}$,$\mathcal{C}_{new} = \mathcal{C}_{new} \bigcup \{$ the counter associated to $t_3\}$,
   replace the counters associated to the removed transitions t_1 and t_2 that appear in all the pre-conditions and post-conditions defined in $\mathcal{T}_0$ with the the counter associated to the transition t_3.

2. Repeat the above step until all reducible pairs in $\mathcal{T}_0$ have been manipulated,

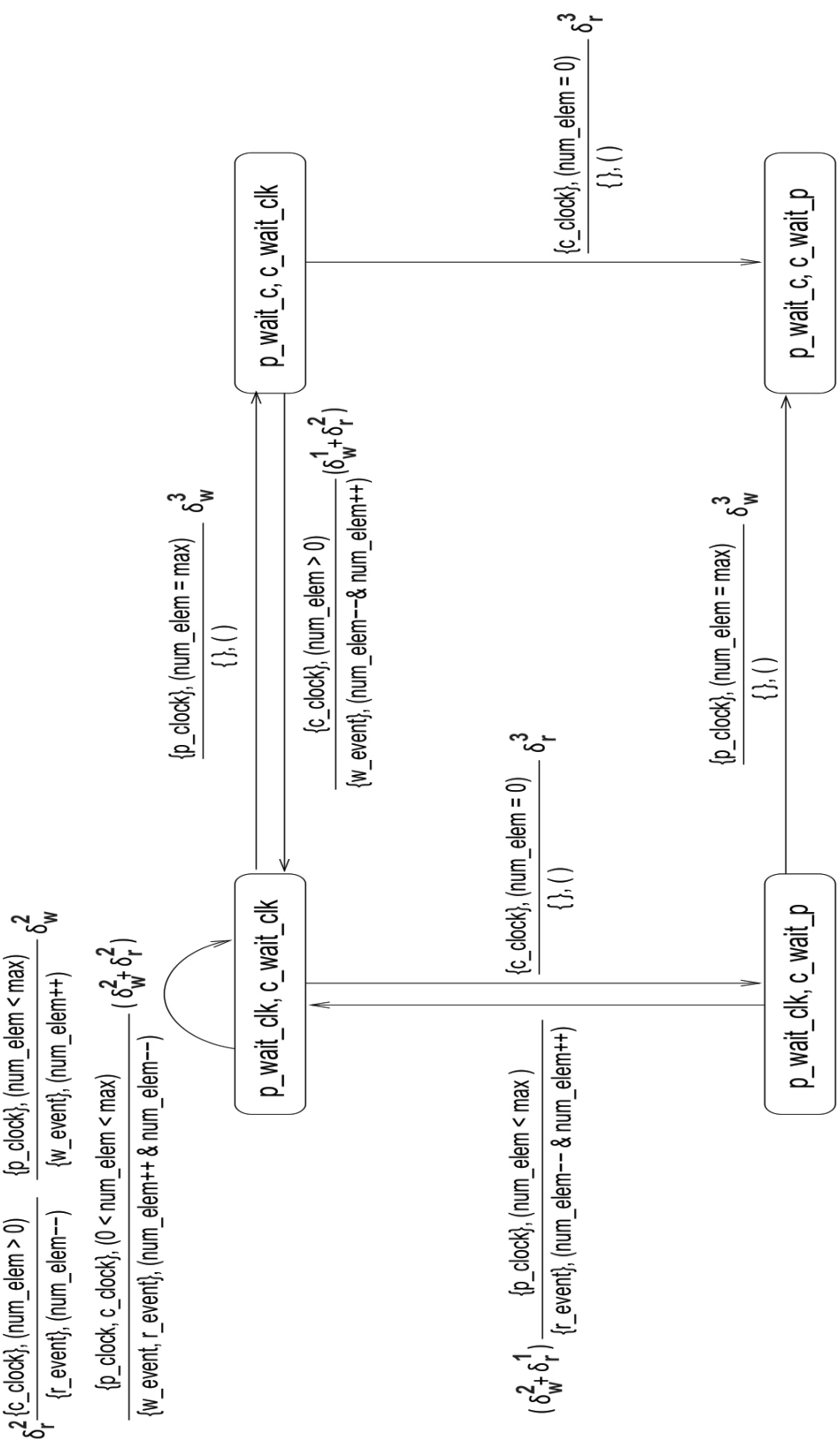


FIGURE 9.3 – *The reduced extended automaton for the FIFO example.*

3. Let $\mathcal{T}_0 := (\mathcal{T}_0/\mathcal{T}_{remove}) \cup \mathcal{T}_{new}$, let $\mathcal{C}_0 := (\mathcal{C}_0/\mathcal{C}_{remove}) \bigcup \mathcal{C}_{new}$ and let $\mathcal{T}_{remove}, \mathcal{C}_{remove}, \mathcal{T}_{new}, \mathcal{C}_{new} := \{\}$.

4. If there are still reducible pairs in $\mathcal{T}_0$, go to the *step 1* and repeat the above procedure, otherwise, let $\mathcal{T}' := \mathcal{T}_0$, and $\mathcal{C}' := \mathcal{C}_0$.

The reduced automaton is $(S, E, \mathcal{T}', \mathcal{C}')$.
In the FIFO example, the reduced automaton is presented in Figure 9.3. Figure 9.3 shows a reduced automaton compared to the automaton in Figure 9.2. But it do more, in fact conditions on counters require further reductions of transitions : this is the main idea behind using the *counters*. In the next subsection, we introduce another extension of the SystemC waiting-state automata using time annotations.

9.2 Extending the SystemC WSA with Time

The SystemC waiting-state automata, as presented in [YZM07], have the notion of global state where information about the system evolution is expressed on transitions. The model expresses also concurrency and synchronization between parallel processes. The SystemC WSA, as presented in the previous section, is also extended with counters that allows to check further properties about the system under study. Indeed, it will be interesting if we add additional information about the **dynamic** behavior of such models which is missing in [YZM07] where all information about time properties get lost. We need to express time information on the SystemC WSA because refining SystemC code with respect to the delta-cycle semantics abstract delta-cycles to untimed atomic transitions. Besides, during SystemC simulation, precise information about execution time are not available.

Time information include the evolution of states and state transitions as well as timing constraints like deadlines, the periodic execution of processes and external event recognition based on time of occurrence. That is why, we propose in this section to extend the work of [YZM07] with time information about the system which is essential for real-time applications that should guarantee correct system behavior. We call the extended automata : the **Timed SystemC waiting-state automata (Timed SystemC WSA)** [HM09].

This is the main contribution to the work in [YZM07] where all information about time properties are not expressed. We propose to annotate each transition with additional infor-

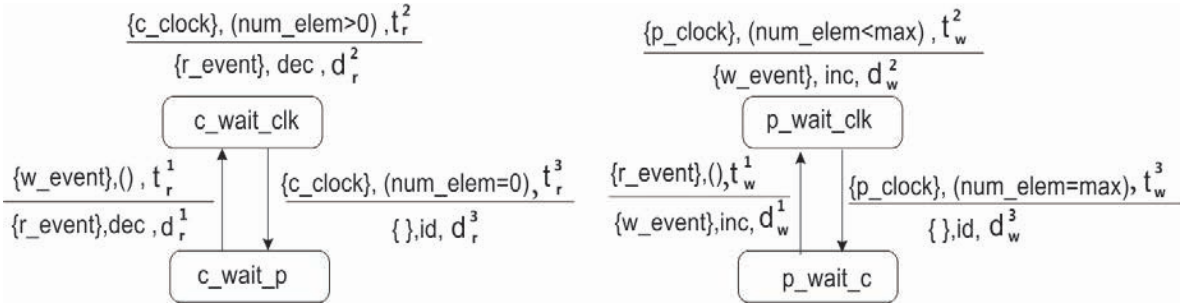


FIGURE 9.4 – *Timed WSA for the producer and the consumer.*

mation about time constraints. More precisely, transitions are timed, i.e. a transition can only be activated if a given time condition is verified and it also defines how long a transition takes to be executed. Verifying temporal properties on a reduced model like the SystemC waiting-state automata is much more easier than on a complex one, since the state space become reduced. Besides, in the SystemC WSA model, transitions represent an abstraction of a set of intermediate internal transitions of each SystemC component within a delta cycle ; i.e, it is mandatory to verify the correctness of the model with respect to strict time constraints at each delta cycle. Moreover, one of the critical problems in hardware designs is the determination of the *WCET (worst-case execution time)*, which is the duration the task could take to execute on a specific hardware platform. Many approaches [CFW01] were developed in this field. We will prove in a next chapter that modeling the hardware using the Timed SystemC waiting state automata gives an exact and an accurate approximation of the WCET. Indeed, we model the processor behavior using the Timed SystemC WSA as presented in [VPM11] and we proceed to an intelligent state fusion as presented in [Ben11].

In this approach, we consider waiting states as instantaneous and eager, i.e. time progress in states is neglectable compared to time in transitions. This allows to express maximal bounds on time progress in transitions. The Timed SystemC WSA is then used to verify the following properties :

- ❖ Time progression and system progression evolve concurrently, waiting states denote the system progression but are timeless.
- ❖ The system can restrict time progress in transitions (by means of delta events and conditional variables).

❖ System transitions can be enabled or disabled by time progress.

In this study, we also consider two types of system communications. Actually timing constraints depend not only on inputs but also on the support of the communication. The support of the communication (the channel in this case) can be either instantaneous, i.e communication between processes occurs at 0 time or with certain latency. Indeed we distinguish :

❑ *Immediate channels*, where communication occurs at 0 time

❑ *Channels with latency*, where communication occurs with certain latency.

9.2.1 Syntax

A Timed SystemC waiting-state automaton over a set $\mathcal{V}$ of variables, is a 5-tuples $A = (S, E, \mathcal{T}, T, D)$, where S is a finite set of states, E is a finite set of events, T is a finite set of transition starting times, D is a finite set of transition durations and $\mathcal{T}$ is a finite set of transitions where every transition is a 8-tuples $(s, e_{in}, p, t, e_{out}, f, d, s')$:

❑ s and s' are two states in S, representing respectively the initial state and the end state of the transition,

❑ e_{in} and e_{out} are two sets of events : $e_{in} \subseteq E, e_{out} \subseteq E$,

❑ p is a predicate defined over variables in $\mathcal{V}$, T and D, i.e., $FV(p) \subseteq \mathcal{V} \bigcup T \bigcup D$, where *FV(p)* denotes the set of free variables in the predicate p,

❑ t is the starting time associated to the transition,

❑ f is an effect function over $\mathcal{V}$,

❑ d is the duration of the transition.

Indeed, a transition in the Timed SystemC WSA is characterized by :

☞ Its functional triggering conditions we are talking about entry and exit conditions p and f.

☞ Input and output events e_{in} and e_{out}.

☞ A time dependent enabling conditions, expressing at which time the transition is possible.

☞ An attribute delay, giving information about the duration that may take each process to change its state.

Let us illustrate this on the FIFO example. Figure 9.4 represents the Timed SystemC waiting-state automata for the producer and the consumer. For instance,t_r^1 represents the time when the transition from c_wait_p state to c_wait_clk state starts, it is actually the time when the consumer is hung up waiting for the producer to write into the buffer. Besides, d_r^1 presents the period of time during which the consumer reads from the buffer. These two values provide meaningful information about the execution time and the behavior of the whole system. For instance, a *read* can only occur if a *write* occurs before it, this property can be represented using durations, i.e., the starting time of transition representing an *r_event* after a *w_event* is always more or equal to the sum of the starting times and durations representing a *w_event*. Formally, it should always hold that :

$$\begin{cases} t_r^1 \geq t_w^1 + d_w^1 \\ t_r^1 \geq t_w^2 + d_w^2 \end{cases}$$

And it is the same for a starting time of transition representing a *w_event*.

$$\begin{cases} t_w^1 \geq t_r^1 + d_r^1 \\ t_w^1 \geq t_r^2 + d_r^2 \end{cases}$$

The previous results are true in the case of immediate channels where communications are instantaneous. These results will be different if we use channels with latency, i.e. we add time constraints due to data transfert in the channel. Indeed, we obtain the following relations :

$$\begin{cases} {t'_r}^1 \geq {t'_w}^1 + {d'_w}^1 \\ {t'_r}^1 \geq {t'_w}^2 + {d'_w}^2 \end{cases} \qquad \begin{cases} {t'_w}^1 \geq {t'_r}^1 + {d'_r}^1 \\ {t'_w}^1 \geq {t'_r}^2 + {d'_r}^2 \end{cases}$$

9.2.2 Symbolic Composition

The symbolic composition of two automata A and A' works similarly as in the case of standard WSA and the last extended WSA.
Our strategy of verifying the SystemC models using the Timed SystemC waiting-state automata requires : (i) to build a minimal-step automaton for every process, (ii) to infer relations over transitions timers (starting time and duration) and (iii) to compose them together to build a larger automaton that can be passed to a model-checker. The symbolic composition of two automata $A(V)$ and $A'(V)$ works similarly as in the case of standard waiting-state automata. What makes the composition a bit more complicated is that values of times of the automata $A(V)$ and $A'(V)$ do not correspond to the values of times of the composed automaton $A(V) \times A'(V)$.

If we consider a transition τ of the automata $A : \tau$ may be composed with many transitions $\{\tau'_1, \ldots, \tau'_n\}$ of the automaton A'. The values of t and d of the transition τ should be represented in the values of times for $(\tau \times \tau'_0)$,..., $(\tau \times \tau'_n)$. We suppose that t^k_τ and d^k_τ denote respectively the starting and the duration associated to the transition $(\tau \times \tau^k)$ in the composed automata, then $t = min_k t^k_\tau$ and $d > \sum_k d^k_\tau$.

We consider a timed automaton as **deterministic** when it does not contain any non-deterministic transition. In the untimed case a deterministic transition has a single start state and from each state, given the next sc_event, the next state is uniquely determined. We want similar criterion for determinism for the timed automata : given an extended state and the next input sc_event along with its time of occurrence, the extended state after the next transition should be uniquely determined.

Algorithm Given two Timed SystemC waiting-state automata $A = (S, E, \mathcal{T}, T, D)$ and $A' = (S', E', \mathcal{T}', T', D')$, over a set $\mathcal{V}$ of variables. The combination of the two Timed SystemC waiting-state automata is still a timed SystemC waiting-state automaton $(S \times S', E \bigcup E', \mathcal{T}'', T'', D'')$ written as $A \times A'$ where $\mathcal{T}''$ is the smallest set of transitions, T'' the associated set of the starting times and D'' is the corresponding set of durations. $\Pi(s, e_{in}, p, t, e_{out}, f, d, s')$ is the set of times of T'' and D'' associated to a transition in $\mathcal{T} \times \mathcal{T}'$. M_t a morphism that maps starting times in $T \times T'$ to T'' and M_d a morphism that maps durations in $D \times D'$ to D'' such that :

- $\Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2) := \{t^*, d^*\} \bigcup \Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)$ and $(s_1, s_1^{'}) \xrightarrow[e_{out}, f, d*]{e_{in}, M_t(p), M_d(p), t^*} (s_2, s_1^{'}) \in \mathcal{T}''$ for every transition $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t} s_2 \in \mathcal{T}$ and for every state $s_1^{'} \in S'$ such that for every transition $s_1^{'} \xrightarrow[e_{out}^{'}, f', d', s_2^{'}]{e_{in}^{'}, p^{'}, t^{'}} s_2^{'} \in \mathcal{T}'$, either $e_{in}^{'} \nsubseteq e_{in}$ or $p \not\Rightarrow p^{'}$,

- $\Pi(s_1', e_{in}', p', t', e_{out}', f', d', s_2') := \{t^*, d^*\} \bigcup \Pi(s_1', e_{in}', p', t', e_{out}', f', d', s_2')$ and $(s_1, s_1^{'}) \xrightarrow[e_{out}', f', d*]{e_{in}', M_t(p'), M_d(p'), t^*} (s_1, s_2^{'}) \in \mathcal{T}''$ for every transition $s_1^{'} \xrightarrow[e_{out}', f', d']{e_{in}', p', t'} s_2^{'} \in \mathcal{T}'$ and for every state $s_1 \in S$ such that for every transition $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t} s_2 \in \mathcal{T}$, either $e_{in} \nsubseteq e_{in}'$ or $p^{'} \not\Rightarrow p$,

- $\Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2) := \{t^*, d^*\} \bigcup \Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)$ and $\Pi(s_1', e_{in}', p', t', e_{out}', f', d', s_2') := \{t^*, d^*\} \bigcup \Pi(s_1', e_{in}', p', t', e_{out}', f', d', s_2')$ and $(s_1, s_1') \xrightarrow[e_{out} \bigcup e_{out}', f \circ f^{'}, d^*]{e_{in} \bigcup e_{in}', M_t(p \wedge p'), M_d(p \wedge p'), t^*} (s_2, s_2') \in \mathcal{T}''$, for every pair of transitions $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t} s_2 \in \mathcal{T}$ and $s_1' \xrightarrow[e_{out}^{'}, f', d']{e_{in}^{'}, p', t'} s_2' \in \mathcal{T}'$.

- According to the transition $(s_1, e_{in}, p, t_{s_1}^{s_2}, e_{out}, f, d_{s_1}^{s_2}, s_2)$, the morphism M_t maps the starting t to the *min* of transitions starting times defined in $\Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)$

$$M(t) \rightarrow min_{t^* \in \Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)} t^*$$

- According to the transition $(s_1, e_{in}, p, t_{s_1}^{s_2}, e_{out}, f, d_{s_1}^{s_2}, s_2)$, the morphism M_d maps the the duration d to the *sum* of durations defined in $\Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)$

$$M(d) \rightarrow d \geq \sum_{d^* \in \Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)} d^*.$$

The algorithm for the symbolic composition of a set of Timed SystemC waiting-state automata is roughly syntactically similar to the algorithm used in Section 9.1. The algorithm used for the SystemC WSA extended with counters is simple : it infers only the relations between the counters used in the minimal-step automata generated for each process and the resulting counters defined for the composed automaton. Thus, it doesn't use further constraints to infer the relations between both sets ; for example constraints defined on the input and the output conditions of the transition. Although, in the case of the Timed SystemC waiting-

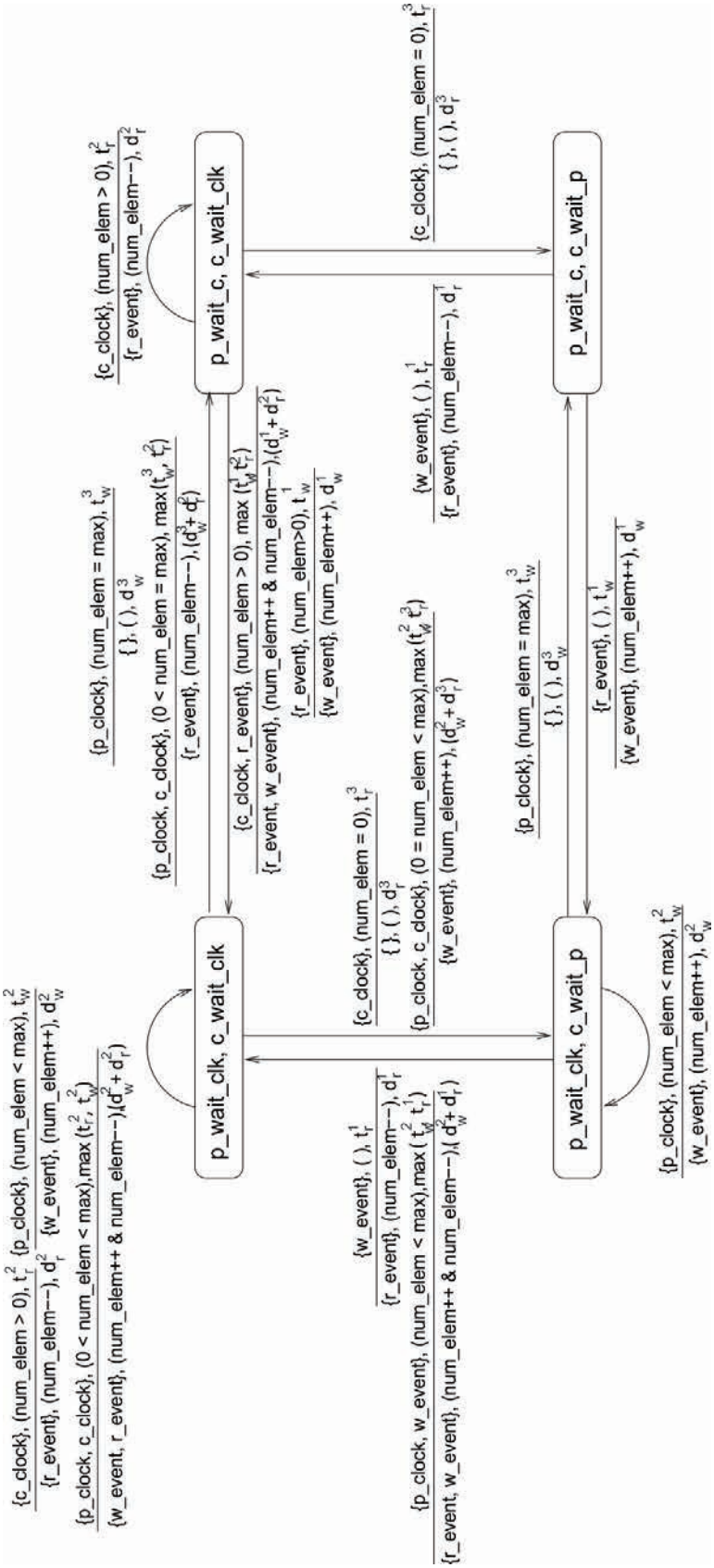


FIGURE 9.5 – *The composed timed WSA for the producer and the consumer.*

state automata, the symbolic composition of the the parallel automata is more intricate since the relations between timing information on the single automata and the composed automaton depend not only on both sets of time parameters but also on the global system time properties. So, in order to decide which transition to keep in the final automaton, we add more constraints on the timing properties of each transition. Each transition, whose time parameters don't respect constraints defined on the global execution time of the system, are automatically removed. Although, in the case of automata extended with counters, those transitions are sometimes preserved.

In conclusion, the algorithm for the symbolic execution of Timed SystemC WSA is more difficult compared to the algorithm used in SystemC WSA extended with counters, because it is based on constraints defined on the time and not only on the functional constraints.

9.2.3 Symbolic Reduction

The process of the symbolic reduction of the Timed SystemC waiting-state automata is similar to that in the previous sections, where the automata were first not annotated with parameters and then they were annotated with counters. In the case of the non-annotated SystemC WSA, the symbolic reduction reduces the set of unfaisable and reducible transitions. Besides, it reduces transitions depending on non environment events as mentioned in Chapter 8. In the case of automata annotated with counters, we use the counters to reduce further transitions (resulting from non-annotated automata) that don't respect the formulas defined over counters as defined in Section 9.1. And now, timing information about transitions, especially the starting time of each transition as well as the duration during which an event is notified, will further enhance the symbolic reduction algorithm.

In fact, we reduce first the subset of time parameters that corresponds to the reduced transitions. The main difference here, is that constraints over time properties help to reduce extra transitions ; i.e, in addition to the reductions that we have applied on non-annotated automata, constraints over time properties requires further reductions. For example, if a transition starts execution at t time, but at least one of the entry conditions (e_{in} and p) is not available at t, then this transition can not be triggered and then removed from the composed automata.

The algorithm for the symbolic reduction of the Timed SystemC waiting-state automata is

different compared to the algorithm used for the symbolic reduction of the automata extended with counters. Hence, in the case of the automata extended with counters, the concatenation of the reducible transitions can be done the same way as in the case of the non-annotated automata. Later, we can add constraints on counters defined on the reducible transitions. But, in the case of the Timed automata, we can not reduce the set of transitions before verifying the time constraints ; i.e, reduction of automata and inferring relations between time constraints are done simultaneously. Thus, the symbolic reduction of the Timed SystemC waiting-state automata is not obvious compared to the symbolic reduction of the automata extended with counters. It requires the joint verification of the constraints defined on the starting time of each transition and its duration, in addition to the detection of the reducible transitions themselves, which requires more refinement of the algorithm used in Section 9.1.

Algorithm Given a Timed SystemC minimal-step waiting state automaton $A = (S, E, \mathcal{T}, T, D)$, where $\mathcal{T}$ has reducible transitions, let $\mathcal{T}_0 := \mathcal{T}$, $T_0 := T$, $D_0 := D$, let $\mathcal{T}_{remove}$, $\mathcal{T}_{new} := \{\}$, T_{remove} ,T_{new}, D_{remove} , $D_{new} := \{\}$.

The following steps define an algorithm to remove all the reducible transitions :

1. for every reducible pair (t_1, t_2) and its contractum t_3, where $t_1, t_2 \in \mathcal{T}_0$, let : $\mathcal{T}_{remove} := \mathcal{T}_{remove} \bigcup \{t_1\}$ and $\mathcal{T}_{new} := \mathcal{T}_{new} \bigcup \{t_3\}$, $T_{remove} := T_{remove} \bigcup \{$ the starting times associated to t_1 and $t_2\}$, $T_{new} := T_{new} \bigcup \{$ the starting time associated to $t_3\}$, $D_{remove} := D_{remove} \bigcup \{$ the durations associated to t_1 and $t_2\}$, $D_{new} := D_{new} \bigcup \{$ the durations associated to $t_3\}$,

   replaces the starting time and the duration associated to the removed transitions t_1 and t_2 that appear in all the pre-conditions and post-conditions defined in $\mathcal{T}_0$ with the new couple of time associated to the transition t_3.

2. repeat the above step until all reducible pairs in $\mathcal{T}_0$ have been manipulated,

3. Let $\mathcal{T}_0 := (\mathcal{T}_0 / \mathcal{T}_{remove}) \cup \mathcal{T}_{new}$, $T_0 := (T_0 / T_{remove}) \bigcup T_{new}$, $D_0 := (D_0 / D_{remove}) \bigcup D_{new}$ and let $\mathcal{T}_{remove}, T_{remove}, D_{remove}, \mathcal{T}_{new}, T_{new}, D_{new} := \{\}$.

4. if there are still reducible pairs in $\mathcal{T}_0$, go to the step 1 and repeat the above procedure, otherwise, let $\mathcal{T}' := \mathcal{T}_0$, $T' := T_0$, $D' := D_0$.

The reduced automaton is $(S, E, \mathcal{T}', T', D')$.

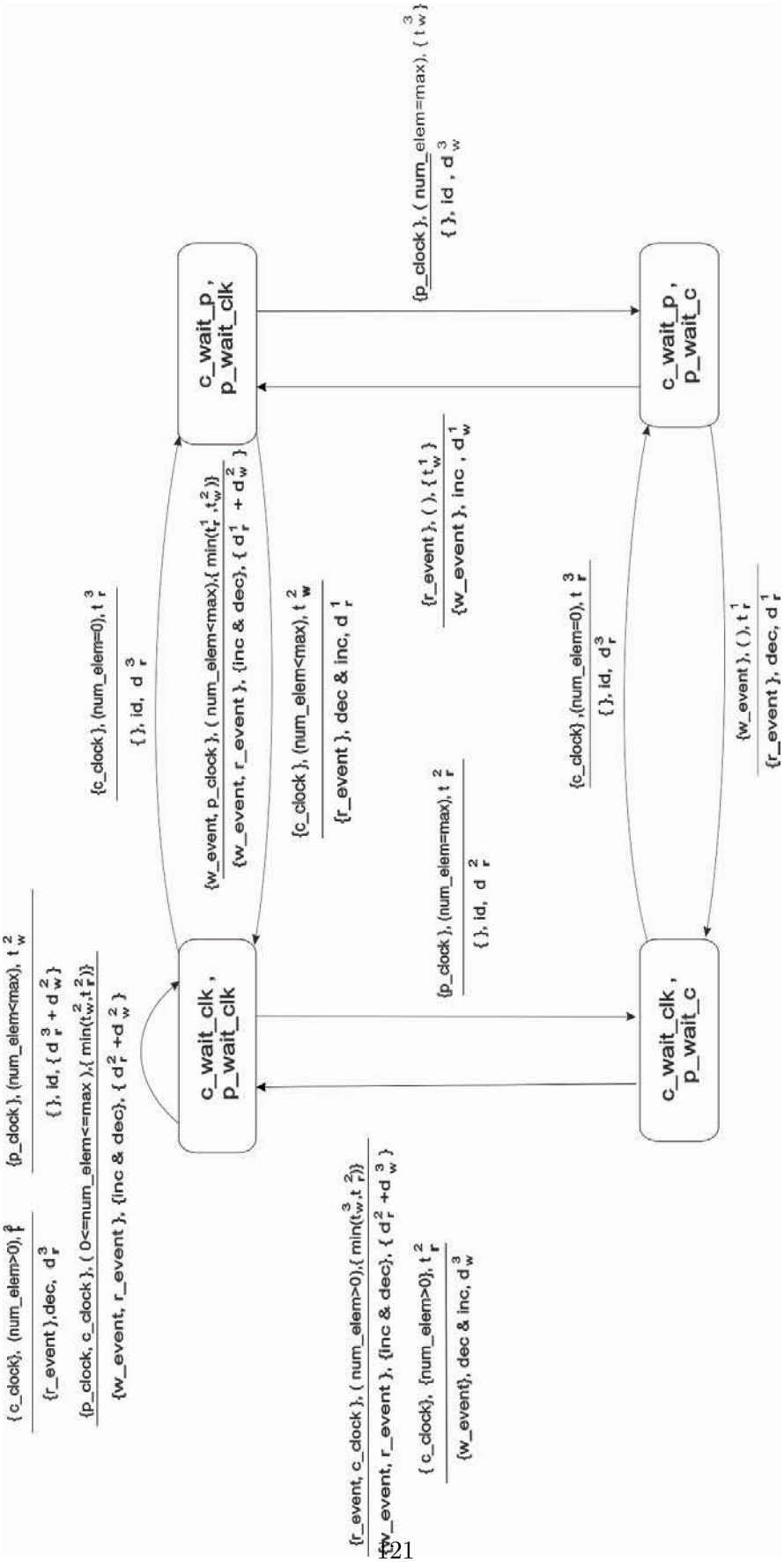


FIGURE 9.6 – *The reduced timed WSA for the producer and the consumer.*

We resume now the reduced automaton in Figure 9.6. As you see, it is similar to the automaton in Figure 8.2, but we add different relations between time parameters defined in the minimal-step automata. For instance, the transition $t = (c\_wait\_p, p\_wait\_clk) \xrightarrow[\{w\_event, r\_event\}, inc \circ dec]{\{w\_event, p\_clk\}, (num\_elem < max)} (c\_wait\_clk, p\_wait\_clk)$ combines the two transitions : $t_1 = (c\_wait\_p) \xrightarrow[\{r\_event\}, dec]{\{p\_clk\}, ()} (c\_wait\_clk)$ and $t_2 = (p\_wait\_clk) \xrightarrow[\{w\_event\}, inc]{\{p\_clk\}, (num\_elem < max)} (p\_wait\_clk)$. Time parameters of t must be defined over those of t_1 and t_2. Using our definition of symbolic composition in Section 9.3.2, the *starting time* of t is the *minimum* of the starting times of t_1 and t_2. But the *duration* of the transition t is the sum of these of t_1 and t_2. We apply the same approach to other transitions to build the automaton in Figure 9.6.

9.3 Extending the SystemC WSA with Counters and Time

In the previous subsections, transitions were first annotated with counters that count the number of times a transition has been activated. Then they were annotated with time information (the starting time and the duration), which make the automaton a well-adapted framework for the verification of reactive hardware/software systems. We have also discussed the effect of that on system properties such as the reduction of state space and the determination of the execution time.

It will be more efficient if we combine both parameters (counters and time) in the same automata. Combining the both information, the time at which a transition is activated, the execution of the transition as well as the number of time a transition has been activated allow to infer precise information about the system time execution ; i.e if we take each process separately and we study how many times it gets into a waiting state and for how much duration, we can compute the lower bound of its execution time. When we compose processes automata together to get a bigger one using the same parameters time and counters, we obtain more information about the model and its execution time and consequently, we are able to verify additional relevant properties and infer new time constraints.

9.3.1 Syntax

An extended Timed SystemC waiting-state automaton over a set $\mathcal{V}$ of variables, is a 5-tuples $A = (S, E, \mathcal{T}, \mathcal{C}, T, D)$, where S is a finite set of states, E is a finite set of events, $\mathcal{C}$ is a finite set of counters, T is a set of transition starting times, D is a set of transition durations and $\mathcal{T}$ is a finite set of transitions where every transition is a 9-tuples $(s, e_{in}, p, t, e_{out}, f, d, s', \delta)$:

- s and s' are two states in S, representing respectively the initial state and the final state of the transition ;
- e_{in} and e_{out} are two sets of events : $e_{in} \subseteq E; e_{out} \subseteq E$,
- p is a predicate defined over variables in V, T and D, i.e, $FV(p) \subseteq V \bigcup T \bigcup D \bigcup \mathcal{C}$,
- t is the starting time associated to the transition that increments each time it is executed,
- f is an effect function over V,
- d is the duration of the transition,
- $\delta \in \mathcal{C}$ is the counter associated to the transition that increments each time it is executed.

The automaton of the producer and the consumer annotated with counters and time is shown in Figure 9.7. we will define now the new relations between both times and counters consi-

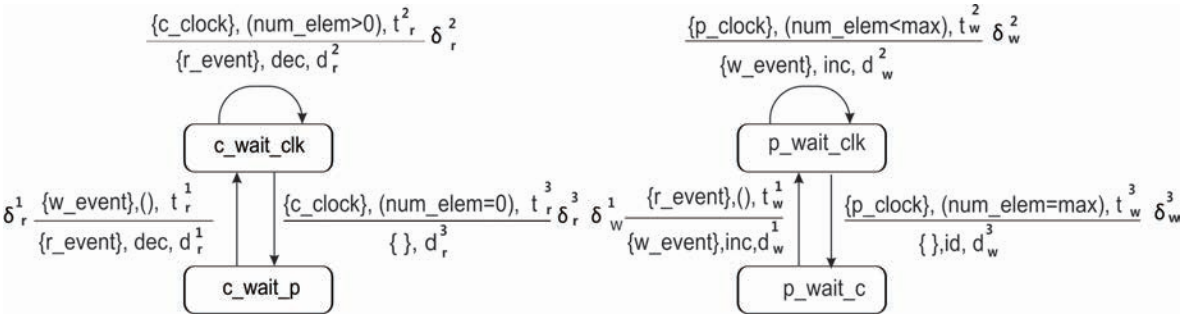


FIGURE 9.7 – *The WSA extended with counter and time.*

dering in addition to that the total execution time. We have usually the same definitions for our parameters, e.g δ_w^3 counts the number of times the transition from state *p_wait_clk* to the state *p_wait_c* is triggered, and t_r^1 represents the time the consumer starts reading. Usually

we have this relation : a read can only occur if a write occurs before it, so when we talk about counters we have these relations :

$$\begin{cases} \delta_w^1 + \delta_w^2 \geq \delta_r^1 + \delta_r^2 \\ (\delta_w^1 + \delta_w^2) - (\delta_r^1 + \delta_r^2) \leq num\_elem \end{cases}$$

Our strategy of verifying temporal constraints in SystemC waiting-state automata is similar as before, we have also these properties that are mentioned before : We can furthermore formulate some properties that may require more precise information, considering both times and counters, i.e we have to verify that the whole reading and writing duration is less than all time execution, so we have the following property :

$$\begin{cases} t_0 + (\delta_w^1 * d_w^1 + \delta_w^2 * d_w^2) + (\delta_r^1 * d_r^1 + \delta_r^2 * d_r^2) \leq T_{exec\_tot} \end{cases}$$

Besides, we consider :

$$\begin{cases} t_r^1 \geq t_w^1 + d_w^1 \\ t_r^1 \geq t_w^2 + d_w^2 \end{cases} \qquad \begin{cases} t_w^1 \geq t_r^1 + d_r^1 \\ t_w^1 \geq t_r^2 + d_r^2 \end{cases}$$

Where, t_0 is the starting time for simulation and T_{exec_tot} is the total execution time.

As we know the first step for verifying SystemC models is to define the minimal step-automaton for every process, and then compose them together so as to get a big automaton for the whole system that will be followed by a reduction procedure. Algorithms for composing and reducing automata , but we consider three essential sets : the set of starting times, the set of durations and the set of counters.

9.3.2 Symbolic Composition

A transition τ of an automaton A may be composed with transitions $\{\tau'_1, \ldots, \tau'_n\}$ of the automaton A', and the values of t, d and δ of the transition τ in A should be represented in the values of times and counters for $(\tau \times \tau'_0)$,..., $(\tau \times \tau'_n)$. If t_τ^k, d_τ^k and δ_τ^k denote respectively the starting time, the duration and the counter associated to the transition $(\tau \times \tau^k)$ in the composed automata, then $t = min_k t_\tau^k$, $d > \sum_k d_\tau^k$ and $\delta = \sum_k \delta_\tau^k$. As the transition predicates (i.e., guard conditions) in the extended timed waiting-state automaton are defined over the

times, counters and system variables, the composition should ensure that these predicates of component automata are properly translated in the composed automaton. In particular, we must replace all the occurrence of a transition starting time, duration and counter from component automata, with the values t, d and δ recently presented.

Algorithm Given two Timed SystemC waiting-state automata extended with counters $A = (S, E, \mathcal{T}, \mathcal{C}, T, D)$ and $A' = (S', E', \mathcal{T}', \mathcal{C}', T', D')$, over a set $\mathcal{V}$ of variables. The combination of the two Timed SystemC waiting-state automata is still a Timed SystemC waiting-state automaton $(S \times S', E \bigcup E', \mathcal{T}'', \mathcal{C}'', T'', D'')$ written as $A \times A'$ where $\mathcal{T}''$ is the smallest set of transitions, $\mathcal{C}''$ is the associated set of counters, T'' the associated set of the starting times and D'' is the corresponding set of durations, $\Pi(s, e_{in}, p, t, \delta, e_{out}, f, d, s')$ is the set of times of T'' and D'' associated to a transition in $\mathcal{T} \times \mathcal{T}'$, M_c a morphism that maps counters in $\mathcal{C} \times \mathcal{C}'$ to $\mathcal{C}''$, M_t a morphism that maps starting times in $T \times T'$ to T'' and M_d a morphism that maps durations in $D \times D'$ to D'' such that :

- $\Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2) := \{t^*, d^*, \delta^*\} \bigcup \Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2)$ and $(s_1, s_1^{'}) \xrightarrow[e_{out}, f, d^*]{e_{in}, M_t(p), M_c(p), M_d(p), t^*, \delta^*} (s_2, s_1^{'}) \in \mathcal{T}''$ for every transition $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t, \delta} s_2 \in \mathcal{T}$ and for every state $s_1^{'} \in S'$ such that for every transition $s_1^{'} \xrightarrow[e_{out}^{'}, f', d', s_2^{'}]{e_{in}^{'}, p', t', \delta'} s_2^{'} \in \mathcal{T}'$, either $e_{in}^{'} \nsubseteq e_{in}$ or $p \not\Rightarrow p^{'}$,

- $\Pi(s_1', e_{in}', p', t', \delta', e_{out}', f', d', s_2') := \{t^*, d^*, \delta^*\} \bigcup \Pi(s_1', e_{in}', p', t', \delta', e_{out}', f', d', s_2')$ and $(s_1, s_1^{'}) \xrightarrow[e_{out}', f', d*]{e_{in}', M_t(p'), M_c(p'), M_d(p'), t^*, \delta^*} (s_1, s_2^{'}) \in \mathcal{T}''$ for every transition $s_1^{'} \xrightarrow[e_{out}', f', d']{e_{in}', p', t', \delta'} s_2^{'} \in \mathcal{T}'$ and for every state $s_1 \in S$ such that for every transition $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t, \delta} s_2 \in \mathcal{T}$, either $e_{in} \nsubseteq e_{in}'$ or $p' \not\Rightarrow p$,

- $\Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2) := \{t^*, d^*, \delta^*\} \bigcup \Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2)$ and $\Pi(s_1', e_{in}', p', t', \delta', e_{out}', f', d', s_2') := \{t^*, d^*, \delta^*\} \bigcup \Pi(s_1', e_{in}', p', t', \delta', e_{out}', f', d', s_2')$ and $(s_1, s_1') \xrightarrow[e_{out} \bigcup e_{out}', f \circ f', d^*]{e_{in} \bigcup e_{in}', M_t(p \wedge p'), M_c(p \wedge p'), M_d(p \wedge p'), t^*, \delta^*} (s_2, s_2') \in \mathcal{T}''$, for every pair of transitions $s_1 \xrightarrow[e_{out}, f, d]{e_{in}, p, t, \delta} s_2 \in \mathcal{T}$ and $s_1' \xrightarrow[e_{out}^{'}, f', d']{e_{in}', p', t', \delta'} s_2' \in \mathcal{T}'$.

- According to the transition $(s_1, e_{in}, p, t_{s_1}^{s_2}, \delta_{s_1}^{s_2}, e_{out}, f, d_{s_1}^{s_2}, s_2)$, the morphism M_t maps the

starting t to the *min* of transitions starting times defined in $\Pi(s_1, e_{in}, p, t, e_{out}, f, d, s_2)$

$$M(t) \rightarrow min_{t^* \in \Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2)} t^*$$

- The morphism M_d maps the the duration d to the *sum* of durations defined in $\Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2)$

$$M(d) \rightarrow d \geq \sum_{d^* \in \Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2)} d^*.$$

- The morphism M_c maps the counter δ to the *sum* of transition counters defined in $Pi(s_1, e_{in}, p, e_{out}, f, s_2, \delta)$

$$M(\delta) \rightarrow \sum_{\delta^* \in Pi(s_1, e_{in}, p, t, \delta, e_{out}, f, d, s_2, \delta_{s_1}^{s_2})} \delta^*.$$

The composed automaton of the FIFo example annotated with both counters and time is shown in Figure 9.8

9.3.3 Symbolic Reduction

The algorithm for symbolic reduction of the composed automata becomes more complicated if we add both time and counters. But the approach is still the same : we define first the set of pairs of reducible transitions and then we replace each pair with only one transition (the *contractum* of both transitions). During the execution of this algorithm, we first reduce unfaisable transitions like in Chapter 8 where automata are not annotated, so we don't have yet conditions about the order of the execution of the transitions neither the number of times a transition can be executed. Next, we need to consider not just the entry-conditions of each transition and its environment events but also the time of occurrence of events and the duration after which the event become visible in the environment. Also, counters inform about variables or events should be modified. Those parameters are important especially in the case of infinite systems where having a finite representation of the system is a major issue.

1. For every reducible pair (t_1, t_2) and its contractum t_3, where $t_1, t_2 \in T_0$, let : $\mathcal{T}_{remove} := \mathcal{T}_{remove} \bigcup \{t_1\}$ and $\mathcal{T}_{new} := \mathcal{T}_{new} \bigcup \{t_3\}$; $\mathcal{C}_{remove} = \mathcal{C}_{remove} \bigcup \{$ the counters associated

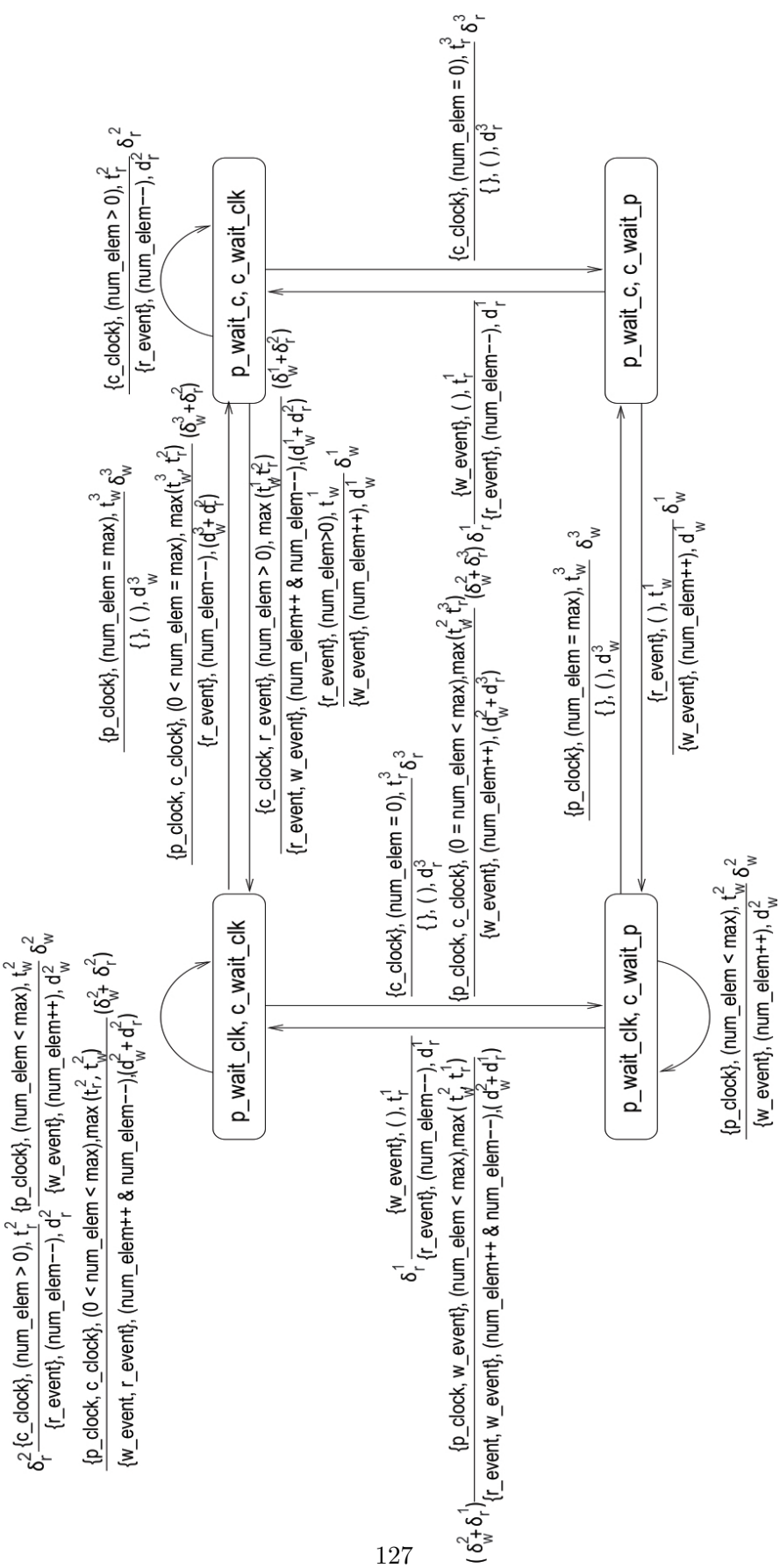


FIGURE 9.8 – *The composed extended automaton.*

to t_1 and t_2} ;$\mathcal{C}_{new} = \mathcal{C}_{new} \bigcup$\{ the counter associated to t_3\} ; $T_{remove} = T_{remove} \bigcup$\{ the starting times associated to t_1 and t_2\} ; $T_{new} = T_{new} \bigcup$\{ the starting time associated to t_3\} ; $D_{remove} = D_{remove} \bigcup$\{the durations associated to t_1 and t_2\} ; $D_{new} = D_{new} \bigcup$\{ the duration associated to t_3\}, replaces the starting times, the durations and the counters associated to the removed transitions t_1 and t_2 that appear in all the pre-conditions and post-conditions defined in T_0 with the new values associated to starting time, duration and the counter associated to the transition t_3.

2. Repeat the above step until all reducible pairs in $\mathcal{T}_0$ have been manipulated ;

3. Let $\mathcal{T}_0 := (\mathcal{T}_0/\mathcal{T}_{remove}) \cup \mathcal{T}_{new}$, $\mathcal{C}_0 := (\mathcal{C}_0/\mathcal{C}_{remove}) \bigcup \mathcal{C}_{new}$, $T_0 := (T_0/T_{remove}) \bigcup T_{new}$, $D_0 := (D_0/D_{remove}) \bigcup D_{new}$ and $\mathcal{T}_{remove}$, $\mathcal{C}_{remove}$, T_{remove}, D_{remove}, $\mathcal{T}_{new}$, $\mathcal{C}_{new}$, T_{new}, $D_{new} := \{\}$.

4. If there are still reducible pairs in $\mathcal{T}_0$, go to the *step 1* and repeat the above procedure ; otherwise, let $\mathcal{T}' := \mathcal{T}_0$, $\mathcal{C}' := \mathcal{C}_0$, $T' := T_0$ and $D' := D_0$.

The reduced automaton is then $(S, E, \mathcal{T}', \mathcal{C}', T', D')$.

The reduced automaton of the consumer and the producer composed together is shown in Figure 9.9.

9.4 Conclusion

In this chapter, we conclude about the global representation of the SystemC waiting-state automata. In fact, we present here different extensions of the model : first, it was extended with counters that are basically used to verify concurrent and distributed systems. Counter automata are used to prove some properties about system behavior such as the reachability property which is difficult to verify in the case of infinite systems. Next, we extend the automata (as an important contribution to the work of [YZM07]) with time annotations [HM09]. We define two parameters : (i) the starting time of the transition and (ii) the duration of the transition. Then we combine the three parameters on the same automata and we prove that we can verify further efficient properties using all the parameters together. We also infer relations between different parameters and we prove that we can enhance both algorithms

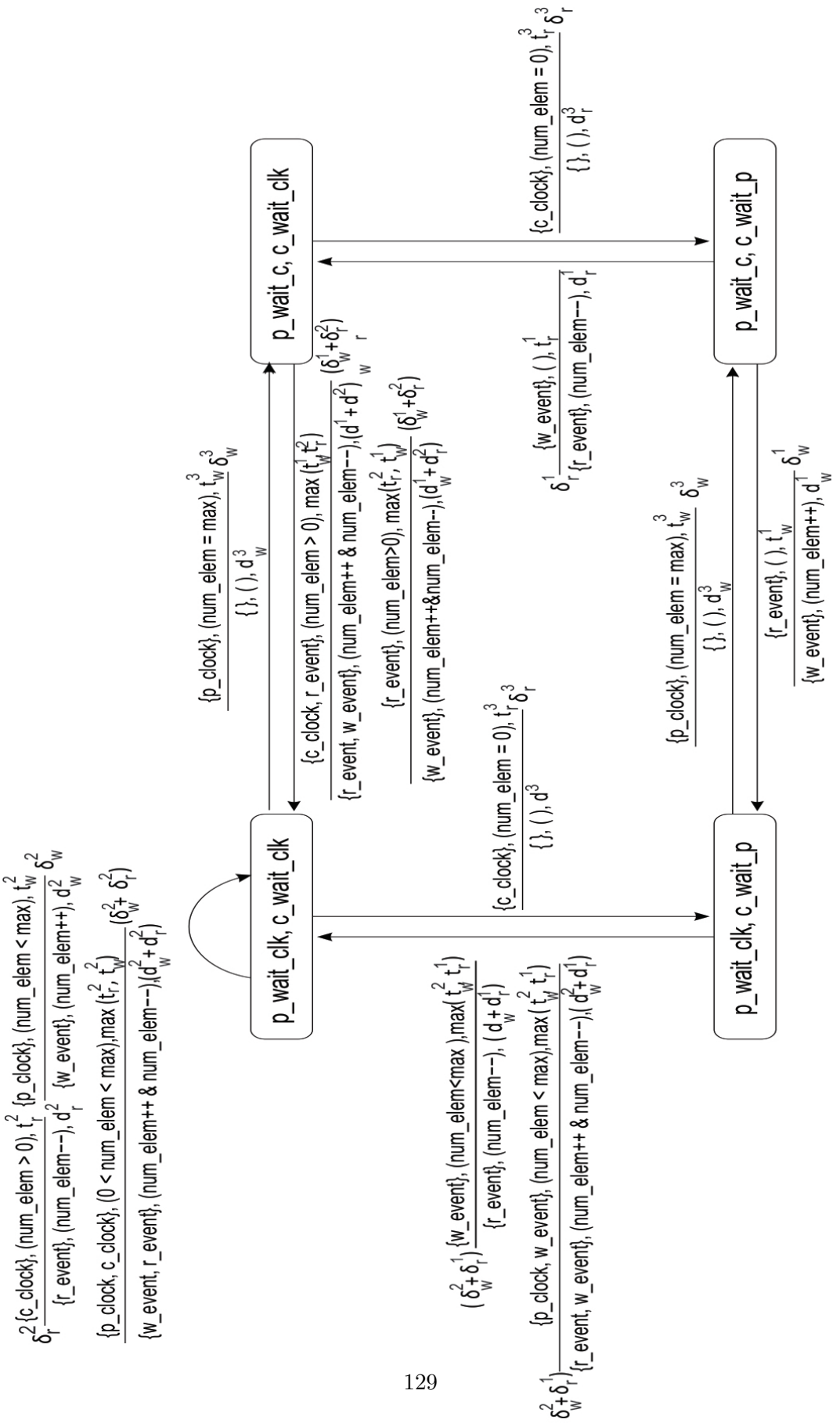


FIGURE 9.9 – *The reduced automaton.*

for the symbolic composition and the symbolic reduction of the global automata and how to make them more efficient.

CHAPTER 10

Summary

In Part *II*, we presented a detailed description of the SystemC waiting-state model as previously presented in papers [YZM07, HM12, NHD11] : First in Chapter 7, we introduced the idea behind the model as it was mentioned by authors in [YZM07]. Indeed, the model gives an abstract representation of the system by reducing its state space modulo the waiting states and with respect to the semantics of the SystemC scheduler at both the delta-cycle level and the system-level.

Next, in Chapter 8, we presented different extensions of the automata using **counters** [YZM07] and **timing properties** [HM09]. Extension with counters was the contribution of work in [YZM07]. But the idea was extended and developed in future work [HM12] : In [HM12], it was mentioned that counters are used basically to verify and avoid unsafe behavior of embedded systems such as the reachability problem and the determinism. Another contribution to work of [YZM07] is to add timing information and constraints on the SystemC waiting-state automata. Timing constraints are essential to verify real-time applications where strict and well-defined deadlines should be respected. So, adding temporal information in addition to counters parameters makes the use of the SystemC WSA model ideal to verify both functional and non-functional properties of real-time systems. The approach was also compared to existing approaches that model and analyze the SystemC language at different

levels of abstractions (another contribution to the work of [YZM07]). It was mentioned that those existing approaches are also modeling SystemC language at well-defined abstraction levels, but they are either restricted to a specific level or they are not usually faithfull to SystemC semantics which was the different case in the approach using the SystemC WSA.

In parallel, we described different algorithms to compose parallel minimal-automata for different components of the whole system and then algorithms how to reduce the composed automaton. These algorithms, as presented first in [YZM07], are also extended in later works [HM12, NHD11]. Those algorithms specify how to generate the set of states and trans itions between them from the initial automata. They also infer the relations between the predicates, the functions associated to each subset of predicates and the relations between different parameters (time and counters).

In the next chapter, we start the first step of the automatic generation of the SystemC waiting-state automata. Indeed, we define how to build automata by separating methods from threads and we illustrate that on some examples.

CHAPTER 11

Mapping SystemC Designs to SystemC WSA

In our approach, we get the description of a system in SystemC. Then we distinguish the constituent components of the system and their communication relation. The processes in a SystemC design, either SC_METHOD or SC_THREAD, build up the components of the system. Each process is modeled as a SystemC waiting-state automaton. The communication and coordination between processes is also mentioned in the operational semantics of SystemC. The behavior of the whole system is compositionally obtained by joining the automata of the processes and their communication scenarios. What we present in this section is also an additional contribution to the work of [YZM07].

11.1 Determining Constituent Components

Each process of a SystemC design (SC_METHOD or SC_THREAD) is considered as a component of the system. The SC_METHOD is an uninterruptible process and has no wait statements. Its automaton must have only one state which is the initial state and the transitions are triggered iff one or more event in the sensitivity list of the process occur or change value. However, the SC_THREAD synchronizes with the environment only through the wait statements and each wait statement represents a state in the abstract model of the process. Transitions are built from the control flow graph of the process.

```
//FlipFlop.h
#include <systemc.h>

SC_MODULE(FlipFlop) {
sc_in<bool> Clk, Reset, DIn;
sc_out<bool> DOut;

void DoFF();

SC_CTOR(FlipFlop) {
SC_METHOD (DoFF);
   sensitive (Reset);
   sensitive_pos(Clk);
// or sensitive_pos<< Clk;
  }
};
```

```
//FlipFlop.cpp
#include "FlipFlop.h"

void FlipFlop::DoFF()
{
  if (Reset)
   {
     Dout= false;
   } else
     if (Clk.event())
      {
       Dout=Din;
      }
   }
};
```

FIGURE 11.1 – *An example of SC_METHOD : a simple FlipFlop Design in SystemC [sys].*

11.2 SystemC WSA of the Components

After determining the components of the system, an abstract automaton is derived for each component. These automata are captured through the *wait* statements in the control flow graph of the related processes. In the following, we show how the automata of the SC_METHOD processes and the automata of SC_THREAD processes are derived respectively.

11.2.1 SystemC WSA of SC_METHOD Processes

The module presented in Figure 11.1 describes a flip-flop with asynchronous reset. When a rising edge occurs on the *Clk* input, *Din* is assigned to *Dout*. Process *DoFF* is a method process, and is called whenever a positive edge occurs on port *Clk* or a transition happens on the input *Reset*, which is an asynchronous reset.

Algorithm

Figure 11.3 shows the algorithm to construct the waiting-state automata of SC_METHOD processes. The waiting-state automaton of an SC_METHOD process has only one state which is the initial state. The transitions are added to the abstract automaton of the process using the paths from and to the initial state of the control flow graph. The occurrence of events on

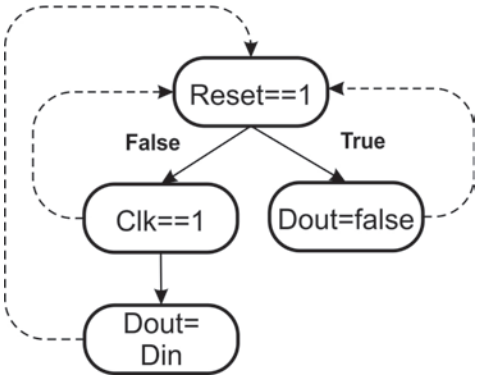


FIGURE 11.2 – *Paths of the Flip Flop SC_METHOD.*

```
Begin
create initial state: S0
For each path in the control flow graph
     For each combination in the sensitivity list
           Add a transition: S0 --> S0
           Condition= condition set of the path
           Action= action set of the path
     EndFor
EndFor
END
```

FIGURE 11.3 – *Algorithm to construct the SystemC waiting-state automata of SC_METHOD Processes.*

each combination of the signals in the sensitivity list of the process, can activate the process independently. Therefore, for each path, at most $2^N - 1$ transitions will be added ; where N is the number of the signals in the sensitivity list of the process. The entry-conditions and the exit-conditions sets of the transitions of the waiting state automaton are equal to the condition set and the action set of the path, respectively.

11.2.2 SystemC WSA of SC_THREAD Processes

The example in Figure 11.4 shows the timer example. The timer process is sensitive to the positive edge of the clock *Clock*, but also depends on the input signal *Start*. The process has just one wait statement, a wait for a positive edge on Clock.

```
//Timer.h
#include <systemc.h>

SC_MODULE(Timer) {
sc_in<bool> Clock, Start;
sc_out<bool> Timeout;
int count;

void RunTimer();

 SC_CTOR(Timer) {
 SC_THREAD (RunTimer);
  sensitive_pos<< Clock;
  count=10;
 }
};
```

```
//Timer.cpp
#include "Timer.h"

void Timer::RunTimer() {
  while (true) {
  wait();
  if (start.read()) {
    count=10;
    Timeout.write(false);
   }
    else {
     if (Count>0) {
       count--;
       Timeout.write(false);}
     else {
       Timeout.write(true);
     }
    }
  {
};
```

FIGURE 11.4 – *An example of SC_THREAD : Timer process written in SystemC [sys].*

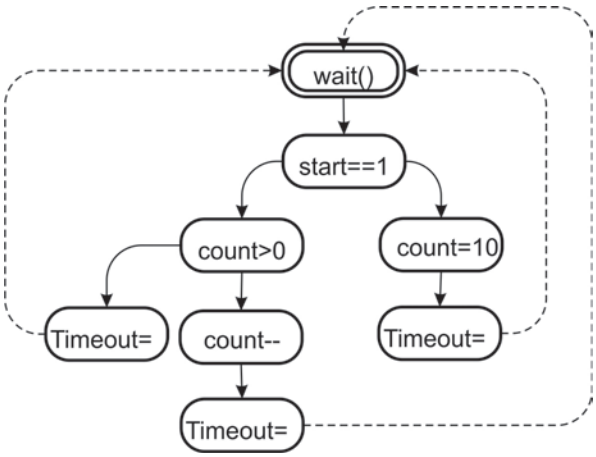


FIGURE 11.5 – *Paths of the Timer-Thread.*

```
BEGIN
create initial state: S0
State: current_state
FOR each path in the control flow graph
  current_state=S0
  FOR each path from current_state to next wait statement
   IF  there exists wait statement after that first wait statement
       create a new state: S
       Add a transition: current_state --> S
       Condition= condition set of the path from current_state to S
       Action= action set of the path from current_state to S
   ELSE
       Add a transition: current_state --> S0
       Condition= condition set of the path from current_state to S0
       Action= action set of the path from current_state to S0
   EndIF
  EndFor
EndFor
END
```

FIGURE 11.6 – *Algorithm to Construct the SystemC waiting-state automata of SC_THREAD Processes.*

Algorithm

Figure 11.6 shows the algorithm to construct the waiting-state automata of SC_THREAD processes. For each path from one waiting state to the next waiting state, there exists a transition. The entry-conditions and the exit-conditions sets of the these transition are equal to the condition and action sets of the corresponding path, respectively. The first transition of the waiting state automaton starts from the first waiting state. For each subsequent waiting state, a state is added from which the second transition in the waiting-state automaton starts. Here, loops are treated like the loops in SC_METHOD processes with a little bit difference, considering wait statements in them. The waiting-state automaton of the Timer Thread is shown in Figure 11.7.

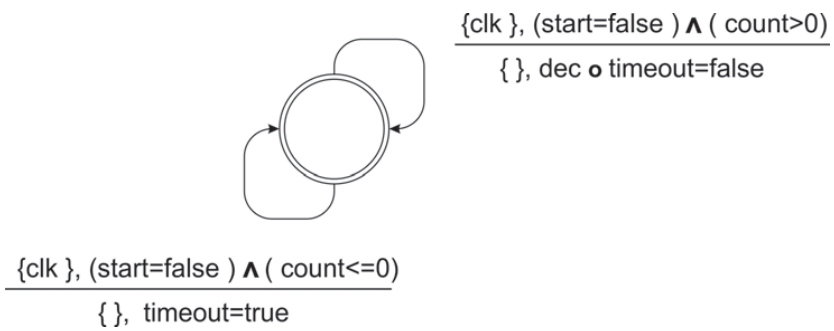


FIGURE 11.7 – *The waiting-state automaton for the Timer Thread*

Part III

Automatic Generation of SystemC Waiting State Automata from SystemC Codes

This Part is composed of Three Chapters ; it describes an automatic approach to build the SystemC waiting-state automata from SystemC codes using different techniques for programs analysis.

CHAPTER 12

Introduction (Global Approach)

In Part *II*, we presented a detailed description of the SystemC waiting-state model and different extension of it. We also defined how to generate automata for both threads and methods processes. We globally defined in Chapter 11 algorithms to generate the wait states for both processes and we illustrate that on some examples.

The SystemC-waiting state automata are transition systems that are manually extracted from SystemC designs[YZM07], so they need first to be automatically generated from the SystemC code and then they should be faithfully conform to the initial system. This is the

main contributions of this thesis : first we have to define an automatic framework to generate the abstract model and next, we should prove the correctness of the model and validate it.

To build the abstract automata, we follow different steps as specified in Figure12.1 : (1) We need to properly write the formal semantics of the SystemC language, we use a small step semantics based on the structural operational semantics of Plotkin [Plo04]. The goal of developing such semantics is (*i*) to provide a complete and unambiguous specification of the language, (*ii*) to execute stepwise the program statement and (*iii*) to detect the effect of that on the global system behavior. We also distinguish between the internal and the global behavior of each SystemC module. All these information are presented in the syntax of the operational semantics of the program. Our semantics capture not only the structure of SystemC components but also the compositional behavior of the communicating components and to do so, we modeled also the behavior of the SystemC scheduler. (2) We proceed to symbolic execution techniques (SE) [Kin76] to generate the control flow graph of the program. We call it a `conjoint symbolic execution` because it combines both symbolic execution and the operational semantics. The main purposes of applying symbolic execution are : first generate different execution traces of the system and second express the program using logic formulas instead of real expressions. This step is a preparatory stage to apply predicate abstraction techniques which represent the next step. (3) We proceed to abstraction techniques, more particularly predicate abstraction (PA)[FQ02] first to infer the relations between the logic formulas generated during the symbolic execution of the parallel automata and second to merge the paths between each two waiting states in the control flow graph. The ultimate goal is to build the SystemC waiting-state automaton from the control flow graph which is annotated with logic formulas defined over global variables and information about the environment events.

Below, we present different steps to automatically generate the SystemC waiting-state automaton from SystemC programs and we illustrate that on different examples.

12.1 Formal Semantics of the Programming Language

The first step in program analysis is to study the formal semantics of the programming language. In fact, defining a proper and detailed description of the program semantics is mandatory in order to give a faithful abstract representation of it. Formal Semantics describes

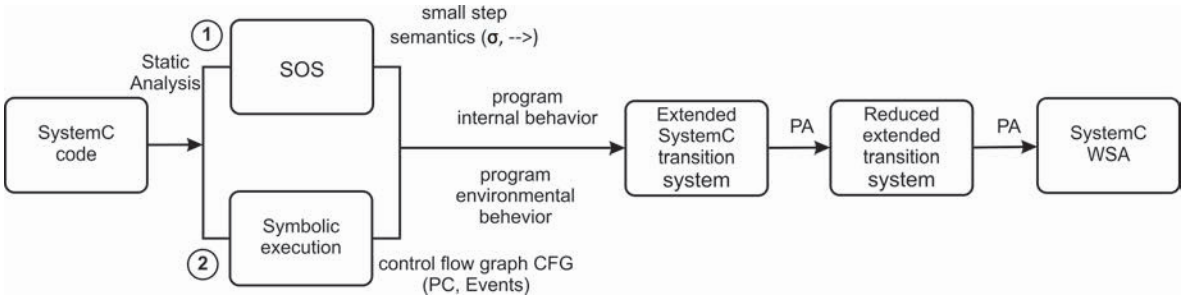


FIGURE 12.1 – *Steps to automatically build the SystemC WSA*

different steps when executing a program in the specific language. This can be shown by describing the relationship between the input and output of a program, or an explanation of how the program will execute on a certain platform, hence creating a model of computation. To study the semantics of SystemC language, we need to properly express how the effect of the computation is produced, i.e, how the program changes from one state to another. Changes may include variables values, the program counter, the environment, etc. We choose to ignore changes like use of registers and addresses for variables. This is the reason why we adopt the operational semantics to describe SystemC programs, more particularly the structural operation semantics (SOS) of Plotkin. They are also called small-step semantics. But before that, let's define first the subset of SystemC language that we will study throughout this thesis.

12.1.1 Subset of SystemC

The goal of this thesis is to propose a global approach to model and then verify SystemC applications from two points of view : first, to prove the correctness of the model and accordingly the SystemC application up to the delta-cycle level where the anomalies generated due to SystemC simulation must be avoided. Second, to prove the correctness of the application on a high level like TLM where details about system behavior are hidden. In this context, SystemC combines both the simulation aspect due to its scheduler and the modeling at different levels of abstractin including the TLM.

Globally, a SystemC component, or module, is an encapsulated piece of code that contains different software structures. Inside such a component, processes may share variables and events in order to synchronize with each other. Communications between modules proceed

mainly by communication channels (for instance bus models). SystemC provides built-in primitive communication channels such as sc_signal to model **hardware** signals at the Register Transfer Level of abstraction. Synchronization associated with the communications is performed by **events** and shared variables inside modules and/or channels.

Throughout this thesis, we consider the previous aspects : communications via channels, events and shared variables, synchronization within a delta cycle, primitives structures, parallel communication.

The previous subset is quit enough to study the correctness of the parallel composition of parallel components, we also capture most information about either low level communication between components using events or the transaction level communication. This restriction is sufficient for us to study SystemC behavior using the SystemC waiting-state automata.

12.1.2 SystemC Operational Semantics

The idea behind the structural operational semantics [Plo04] is to describe how to execute the program and not merely what the results of execution are. More precisely, we are interested in how the states are modified during the execution of the statements.

The operational semantics represent the program as a transition system. The transition system is a structure $< \Gamma, \rightarrow >$ where Γ is a set of elements, it is also called **configuration** and $\rightarrow \subseteq \Gamma \times \Gamma$ is a binary relation called **the transition relation**.

Example To illustrate how the SOS are executed, we take as an example the IMP language as presented in [Win93], it is a small imperative language. We suppose the syntax below of the language :

- Variables $x \in Var = \{x_0, x_2, ..., x_n\}$.
- Numbers : m, n are meta variables over $\mathbb{Z}$.
- Arithmetic expressions **AExp** : $a_0, a_1, ...$
- Boolean expressions **BExp** : $b_0, b_1, ...$
- The configuration Conf : $c_0, c_1, ...$

Variables	$\frac{}{<x,\sigma> \rightarrow a\sigma[x]}$
Arithmetic Reductions	$\frac{}{<n \oplus m,\sigma> \rightarrow a\ p}$ where $p = n \oplus m$
Other arithmetic expressions	$\frac{<a_1,\sigma> \rightarrow_a <a_1',\sigma>}{<a_1 \oplus a_2,\sigma> \rightarrow a\ a_1' \oplus a_2}$ $\frac{<a_2,\sigma> \rightarrow_a <a_2',\sigma>}{<n \oplus a_2,\sigma> \rightarrow a\ n \oplus a_2'}$
Assignement	$\frac{}{<x:=n,\sigma> \rightarrow <skip,\sigma[x/n]>}$ $\frac{<a,\sigma> \rightarrow_a a'}{<x:=a,\sigma> \rightarrow <x:=a',\sigma>}$
Sequences	$\frac{<c_0,\sigma> \rightarrow <c_0',\sigma'>}{<c_0;c_1,\sigma> \rightarrow <c_0';c_1,\sigma'>}$ $\frac{}{<skip;c_1,\sigma> \rightarrow <c_1,\sigma>}$
If statements	$\frac{<b,\sigma> \rightarrow_b b'}{<\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma> \rightarrow <\textbf{if } b' \textbf{ then } c_0 \textbf{ else } c_1,\sigma>}$ $\frac{}{<\textbf{if true then } c_0 \textbf{ else } c_1,\sigma> \rightarrow <c_0,\sigma>}$ $\frac{}{<\textbf{if false then } c_0 \textbf{ else } c_1,\sigma> \rightarrow <c_0,\sigma>}$
While statements	$\frac{}{<\textbf{while } b \textbf{ do } c,\sigma> \rightarrow <\textbf{if } b \textbf{ then } (c;\textbf{while } b \textbf{ do } c) \textbf{else skip},\sigma>}$

TABLE 12.1 – SOS for IMP language

$$
\begin{array}{llll}
\text{(AExp)} & \text{a} & ::= & n|x|a_0 \oplus a_1 \\
\text{(BExp)} & \text{b} & ::= & true|false|a_0 \odot a_1|b_0 \oslash b_1|\neg b \\
\text{(Conf)} & \text{c} & ::= & skip|x := a|c_0; c_1|\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2|\textbf{while } b \textbf{ do } c \\
 & \oplus & ::= & +|*|- \\
 & \odot & ::= & <|>|\geq|\leq|= \\
 & \oslash & ::= & \wedge|\vee
\end{array}
$$

We use the notation σ to represent the **store**, it is a function over variables such as : $\sigma : Var \rightarrow \mathbb{Z}$. The SOS of the IMP language use a set of rules to define the set of configurations as below :

$$< c, \sigma > \rightarrow < c', \sigma' > \tag{12.1}$$

The notation 12.1 is used to describe how a statement changes the configuration from c to c' in one step. To describe different syntactic configurations of IMP language, we use the following notations :

$$
\begin{array}{ll}
\rightarrow : & (Conf \times \Gamma) \rightarrow (Conf \times \Gamma) \\
\rightarrow_a : & (AExp \times \Gamma) \rightarrow AExp \\
\rightarrow_b : & (BExp \times \Gamma) \rightarrow BExp
\end{array}
$$

12.2 Symbolic Execution

The main idea behind symbolic execution [Kin76, Dar88] is to use symbolic values, instead of actual data to represent the values of program variables as well as the input values. As a result, the output values computed by a program are expressed as a function of symbolic values. Evaluation of assignments is done naturally ; the left-hand sided variable receives the resulting symbolic expression, which should be a polynomial.

Evaluation of alternatives is a bit more complicated. It requires that a *path condition* PC -a Boolean expression over the symbolic inputs- is added to the execution state. The path condition PC is a (quantifier free) boolean formula over the symbolic inputs. It accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated execution path.

At program start, each symbolic execution begins with PC initialized to **true**. When encountering an alternative, evaluation first starts with the evaluation of the associated boolean expression by replacing variables by their values. Since the values of variables are polynomials over the symbols, the condition is an expression of the form : $P > 0$, where P is a polynomial. Call such an expression R. Then we can have three cases :

- $PC \supset R$ and $PC \not\supset \neg R$: In this case, the expression is always true, the execution continues with the conditional code sequence.
- $PC \supset \neg R$ and $PC \not\supset R$: In this case, the expression is always false, the execution continues with the **else** code sequence if an **else** block is available or simply ignore the conditional code sequence.
- Otherwise, the boolean condition may be **true** or **false**. In this case, we split the path condition into two paths conditions $PC_{true} = PC \wedge R$ and $PC_{false} = PC \wedge \neg R$ and we continue the concurrent execution of the condition code sequence with PC_{true} and the **else** code sequence or the code located after the conditional code sequence with the path condition PC_{false}.

Example If we consider the program below :

```
1 ASSUME(true);
2 DECLARE X, Y INTEGER;
```

```
3 IF  (X < 0)
4 THEN Y := –X
5 ELSE Y := X
6 RETURN(Y) ;
```

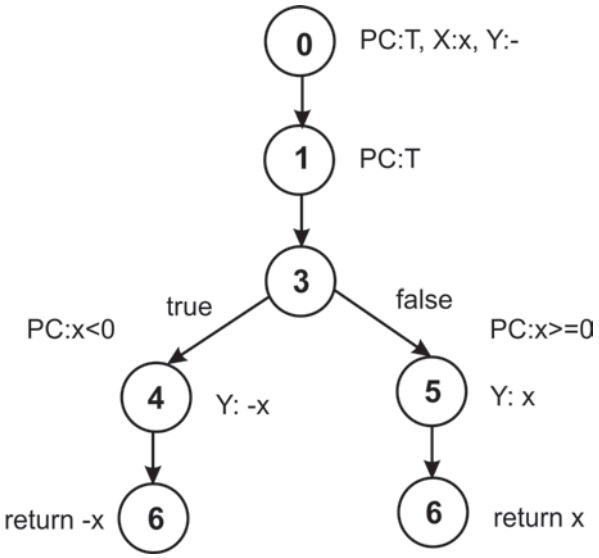


FIGURE 12.2 – *Symbolic execution of a the example*

As mentioned in Figure 12.2, symbolic execution generates all possible execution of the program with accumulation of formulas.

In our approach, we use symbolic execution to generate all possible execution traces using symbolic values instead of real ones. Besides, SE is one of the common methodologies in static program analysis. It generates a *control flow graph (CFG)* : an abstract representation of the behavior of the program. Then, we use abstraction techniques like *predicate abstraction* to analyze and optimize the graph (as we will present in next Section).

12.3 Abstract Analysis and Traces Merging

Abstraction techniques like *predicate abstraction* [FQ02], which is a special variant of *abstract interpretation* [CC77, CC92], are widely used for semantics based static analysis of software. These techniques are based on two main key-concepts : the correspondence between the concrete and the abstract semantics through the *Galois connections*, and the feasibility of a fixpoint computation of the abstract semantics, through the fast convergence of *widening*

operators.

In the previous section, we enumerate the usefulness of symbolic execution to generate the set of the execution traces by generating the control flow graph of the SystemC program. However, the symbolic execution is itself not approximative, but as precise as possible (which corresponds to generating abstract formulas instead of real ones). Instead, the necessary approximation is performed by explicit *abstraction* operations, which make use of an arbitrary finite set of predicates over the variables of the program. Let us briefly recall some basic definitions.

12.3.1 Galois Connection

A Galois connection is a pair of functions (α, γ) defined over a set of partially ordered sets (Posets). The abstraction function is denoted α and the concrete function is denoted γ.

Definition 1 Let $\langle C, \leq \rangle$ and $\langle D, \sqsubseteq \rangle$ be two posets, and consider two monotonic functions $\alpha : C \to D$ and $\gamma : D \to C$. The tuple $G_{CD} = (\gamma_{DC}, C, D, \alpha_{CD})$ is a Galois connection if :

$$\forall X \in C \; and \; \forall Y \in D : \alpha(X) \sqsubseteq Y \Rightarrow X \leq \gamma(Y)$$

In a Galois connection or G_{CD}, the functions γ_{DC} and α_{CD} are called the *concretization* and the *abstraction* function, respectively. The following are well-known properties of these functions (see [CC92]).

Lemma 1 $\langle C, \leq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle D, \sqsubseteq \rangle$ is a Galois connection if and only if :

- $\gamma_{DC} \circ \alpha_{CD}$ *is extensive* : $\forall c \in C, c \leq \gamma_{DC}(\alpha_{CD}(c))$;
- $\alpha_{CD} \circ \gamma_{DC}$ *is reductive* : $\forall d \in D, \alpha_{CD}(\gamma_{DC}(d)) \sqsubseteq d$

Lemma 2 LetGG_{CD} be a Galois connection,

- if α_{CD} and γ_{DC} form a Galois connection, then one of the two functions determines the other one. More precisely, for $d \in D$, $\gamma_{DC}(d) = \sqcup_C \{c \in C \mid \alpha_{CD}(c) \sqsubseteq_D d\}$, and similarly, for $c \in C$,$\alpha_{CD}(c) = \sqcap_D \{d \in D \mid c \sqsubseteq_C \gamma_{DC}(d)\}$. Each function is called the adjoint of the other one.

- $\alpha_{CD} \circ \gamma_{DC} \circ \alpha_{CD} = \alpha_{CD}$, *and* $\gamma_{DC} \circ \alpha_{CD} \circ \gamma_{DC} = \gamma_{DC}$.

12.3.2 Fixed points Computation

Traditionally, in abstract interpretation, abstract and concrete semantics are defined as the computation of the fixpoint of monotonic functions. The goal behind fixpoint computing is to prove that the abstract semantics are faithful to the concrete semantics and vis versa, i.e, we verify that the concretization of the abstract results is an over-approximation of the concrete semantics.

We use the following two lemma to prove that a fixpoint exists in the abstract semantics.

Lemma 3 (Kleene [kle38]) Let $\langle L, \sqsubseteq, \sqcup \rangle$ and $\langle \overline{L}, \overline{\sqsubseteq}, \overline{\sqcup} \rangle$ be two complete lattice and consider two monotone functions $F : L \to L$ and $\overline{F} : \overline{L} \to \overline{L}$ respecting respectively $\sqsubseteq$ and $\overline{\sqsubseteq}$. We consider α a morphism on the concatenation such that : $\alpha \circ F \overline{\sqsubseteq} \overline{F} \circ \alpha$. Let $a \in L$ a *prefix point* in F, then the following property is true :

$$\alpha \left(lfp_a^{\sqsubseteq} F \right) \overline{\sqsubseteq} lfp_{\alpha(a)}^{\overline{\sqsubseteq}} \overline{F}.$$

Lemma 4 (Knaster-Tarski [Tar55] Let $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ be a complete lattice and $\varphi : L \to L$ be a monotone function. φ has a fixpoint :

$$\begin{aligned} \mathbf{lfp}\varphi = & \quad \sqcap \mathbf{postfp}(\varphi) \\ & = \sqcap \{x \in L | \varphi(x) \sqsubseteq x\} \end{aligned}$$

We do the same for the *greatest fixpoint* :

$$\begin{aligned} \mathbf{gfp}\varphi = & \quad \sqcup \mathbf{prefp}(\varphi) \\ & = \sqcup \{x \in L | x \sqsubseteq \varphi(x)\} \end{aligned}$$

12.3.3 Widening Operators

In Abstract Interpretation, the collecting semantics of a program is expressed as a least fixpoint of a set of equations. The equations are solved over some abstract domain that captures the property of interest to be analyzed. Typically, the equations are solved iteratively ; that is, successive approximations of the solution is computed until a fix-point is reached. However,

for many useful abstract domains, such chains can be either infinite or too long to let the analysis be efficient. To make use of these domains, abstract interpretation theory provides very powerful tools, the widening operators, that attempt to predict the fix-point based on the sequence of approximations computed on earlier iterations of the analysis on a cpo or on a (complete) lattice. The degradation of precision of the solution obtained by *widening* can be partly restored by further applying a *narrowing* operator.

Definition 2 (Widening [CC92]). Let $(P, \leq)$ be a poset. A set-widening operator is a partial function $\bigtriangledown : A \to A$ such that :

i) Covering : Let S be an element of P. If $\bigtriangledown(S)$ is defined, then $\forall x \in S$, $x \leq \bigtriangledown(S)$.

ii) Termination : For every ascending chain $\{x_i\}_{i \geq 0}$, the chain defined as :
$y_0 = x_0, y_i = \bigtriangledown(\{x_j \mid 0 \leq j \leq i\})$.
is ascending too, and it stabilizes after a finite number of terms.

The definition above has been used recently in [DP90, D'S06], for fix-point computations over sets represented as automata in a model checking approach.

Example Consider a lattice of intervals [Cor08] $L = \{\bot\} \cup \{[l, u] \mid l \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$, ordered by : $\forall x \in L, \bot \leq x \; and \; [l_0, u_0] \leq [l_1, u_1] \; if \; l_0 \leq l_1 \; and \; u_0 \leq u_1$. Let k be a fixed positive integer constant, and I be any set of indices. Consider the threshold widening operator

$$\bigtriangledown^k(\{\bot\}) = \bot$$

defined on L by : $\bigtriangledown^k(\{\bot\} \cup S) = \bigtriangledown^k(S)$ where :

$$\bigtriangledown^k(\{[l_i, u_i] : i \in I\}) = [h_1, h_2]$$

$$h_1 = min\{l_i : i \in I\} \; if \; min\{l_i : i \in I\} > -k, \; else \; -\infty$$

$$h_2 = max\{u_i : i \in I\} \; if \; max\{u_i : i \in I\} < k, \; else \; +\infty$$

Observe that for all k, $\bigtriangledown^k$ is associative, and monotone. However, it is not reflexive. For instance, we get $\bigtriangledown^7(\{[-8, 4]\}) = [-\infty, 4]$

12.3.4 Traces merging

Traces merging encodes first the set of feasible traces according to the semantics of the programming language and then transfoms them into paths with respect to the control flow graph. We will use traces merging to generate the set of transitions in the SystemC waiting-state automata from the control flow graph generated during symbolic execution.

During symbolic execution : the program begins in some initial state, and each execution step transforms the current state into a new state until it reaches (or not) a final state. We note S the set of states that might be encountered during executions. An execution of a program generates a countably infinite (or finite) sequence of states called an execution trace.

Definition 3 We consider an automaton $A = (S, \mathcal{T})$, where S is a finite set of states and $\mathcal{T}$ a finite set of transitions. We will formally write a trace π as a sequence of states such as : $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_n} s_n$ Where $s_0 \in S$ and $s_n \in S$ represent respectively the first state and the final state in the trace π and $t_0 \in \mathcal{T}$ and $t_n \in \mathcal{T}$.

The notion of trace can be adapted to display the set of inputs or events that activate the transition from one state to another in the trace. We consider a sequence, $\pi = s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \ldots \xrightarrow{l_n} s_n$, where each label l_i triggers a transition from s_i to s_{i+1}. The label can be either an event e_i, a predicate p_i or an effect function (or an action) f_i.

The program might be executed with different start states, s_0, and a non-deterministic program might generate different execution traces from the same initial state s_0 to the same final state s_n in the case of a finite system. For this reason, one execution might generate a set of traces. Or, the traces generated by different executions can be grouped into one trace set.

To remove the previous ambiguity we resort to traces merging ; this step consists in abstracting the set of traces that starts from the same intial state and leads to the same final state into one transition. That is why, we define a galois connexion (α, γ) that transforms the set of concrete traces into a set of abstract traces. We consider the following definition :

Definition 4 We consider respectively Σ^C and Σ^A the set of the concrete and the abstract traces. We relate abstract traces to concrete traces by defining the galois connexion as below :

$$\langle \Sigma^C, \subseteq \rangle \underset{\gamma^{Trace}}{\overset{\alpha^{Trace}}{\rightleftarrows}} \langle \Sigma^A, \sqsubseteq \rangle$$

- $\gamma^{Trace}(\pi^a) = \{\pi^c \in \Sigma^c | \gamma^{Trace}(\pi^a) = \pi^c\}$
- $\alpha^{Trace}(\pi^c) = \bigcup\{\alpha(\pi^c) | \pi^c \in \Sigma^C\}$

Example In paper [MHP09], authors propose a counterexample-guided abstraction refinement approach. The approach refines an over-approximation of the set of `possible traces` generated from the control flow graph. Each refinement step introduces a `finite automaton` that spot a set of infeasible traces. They use `interpolants` to extract automata that they call `interpolant automata`. The idea of the approach over this example is to abstract the set of feasibale traces from the set of all the traces generated in the control flow graph.

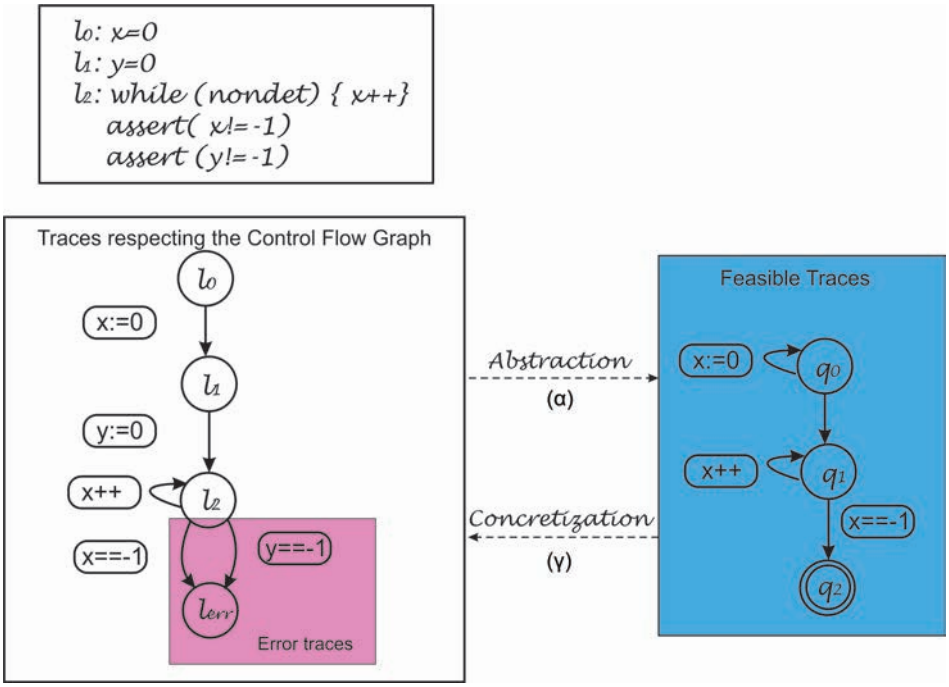


FIGURE 12.3 – *An Example with Traces Merging [MHP09]*

CHAPTER 13

Formal Semantics of SystemC

In this chapter, we define the behavioral semantics of a subset of SystemC language. This analysis can be extended to handle all components of SystemC language. The goal of developing SystemC formal semantics is to provide a complete and unambiguous specification of the language. It also contributes significantly to information sharing, to description portability, and to integration of various applications in simulation, synthesis, and formal verification.

Over the last ten years or so, research in formal semantics in electronic design community mainly focused on Verilog, VHDL and SystemC. Quite often, their definitions were based on

Abstract State Machine (ASM) specifications [BS03, AGT04] or on the *Denotational Semantics (DSs)* [Sal03]. But, it was generally believed that SOS provide more intuitive descriptions especially to describe the dynamic behavior of the system than ASM specifications and denotational semantics.

13.1 Motivations and Related Works

To formalize concurrency on programming languages such as SystemC, we need to define formally its semantics. This problem has already been studied for other languages such as Java [HP00, Hav00], with the aim of performing formal verification.

Several attempts were made in order to formalize SystemC semantics, some of them target the RTL (Register Transfer Level) subset of SystemC and lately few works studied the TLM subset of SystemC. But none of them stresses on the correctness of SystemC at the delta-cycle level. There is only one interesting approach that studies the semantics of SystemC at the RTL, the TLM and the delta-cycle level : the approach of R.K. Shyamasundar, F. Doucet,R. Gupta, and I.H. Kruger in [RSK07]. Authors in [RSK07] propose a global framework that presents the behavioral semantics of SystemC. Those semantics succinctly capture the reactive features, clock and time references of SystemC. Their semantics allow the specification of a network of synchronous and asynchronous components communicating through either high-level transactions or low-level communications. Our semantics are based on those in [RSK07], but we do more here. Indeed, we use first different notations to express transformations on transitions : we express transformations on environment events, on global variables and on request updates for channels. The latter notation was not expressed in [RSK07], although it is obvious and essential to study and manage updates on channels. Second, during the definition of the simulation semantics of SystemC scheduler, we distinguish between the three phases of the simulation semantics of the scheduler which was not addressed in [RSK07].
Among existing works, we can identify two different approaches :

The first one is to provide a *simulation semantics*, that have to take into account the scheduler itself and model its different simulation phases. In work of [WRM01, AGT04], SystemC semantics are expressed in terms of abstract state machines. A separate process is used to represent the scheduler. Unfortunately, this is not conform to the cooperative aspect of SystemC scheduler since processes are executing concurrently. In [Sal03], authors

use denotational semantics to formalize SystemC Models and model separately the scheduler. However, this latter semantics does not allow expressing any control-flow between states, which is quite limiting. These previous works have in common two limitations : First, they rely on the assumption that components communicate through the use of sc_signal channels when modeling the scheduler. Then, the target formalism that is used does not have associated concrete tools. It is therefore impossible to check on real and complex examples that the given semantics does indeed correspond to SystemC. Therefore, we cannot say that those semantics are as precise as enough to SystemC since they are not applied on real examples. Besides, they are not faithful enough to the concrete semantics of the scheduler. In[WMR03], Muller, et al. define the semantics of SystemC using distributed abstract state machines.

The second approach extends the first one to study SystemC semantics at the transactional level. We may mention the work of [MFM06, KMS06, NH06, Moy05] and later the work of [PXN06, RSK07]. Although those works solve the problem of modeling the scheduler independently from the implementation and study the properties of the transactional level, but they still luck of granularity since they don't stress on the delta-cycle semantics of SystemC. This represents the main contribution of the approach using the SystemC waiting-state automata. In fact, the SystemC waiting-state automata [YZM07, HM12] allows : (i) to decompose the SystemC code into communicating models, each model presents a process in the SystemC structure, this model conforms to SystemC semantics at both the delta-cycle level and the TLM, (ii) to model the behavior of the whole system by composing the elementary automata. The waiting-state automaton solves the problem of state explosion since it considers only states when a component is communicating with its environment, i.e, when it is visible by other components and it ignores intermediate transitions that describe the local behavior of the component.

We mention some recent works that study and model the formal semantics of SystemC [DTS08, PHG08, DGeD10, ACR10]. In [DTS08], authors define a trace semantic for SystemC covering abstract models. [PHG08] uses an existing representation for system analysis to model/verify SystemC semantics based on the semantics of Uppaal models [JBU]. Work of [DGeD10] propose a fully automatic SystemC-to-C transformation, it uses C assertions and finite state machines to verify properties related to Transaction-Level Modeling (TLM). Finally, [ACR10] transforms SystemC into a sequential C program and then proceeds to

lazy abstraction techniques to deal with multi-threaded software and cooperative scheduling. Those works have good results in term of SystemC verification and applying static analysis techniques to verify the program but they still luck of innovation. Although, in this present work (*i*) we define the formal semantics of SystemC with respect to its execution semantics and to the semantics of the SystemC waiting-state automata, (*ii*) we also use new techniques for program analysis at different stages of the approach in order to validate and automatically abstract the formal model from the SystemC description.

We have presented several works covering the formalization of SystemC. However, they are limited either to the RTL subset of the language or the TLM, and none of them studied the delta-cycle level. We also need a mean to check that the semantics are faithful to SystemC. This point is partly covered by the latest works, which analyze the SystemC code, generate the model and connect it to existing verification tools : this provides a mean to check that the given semantics is actually related to the SystemC simulation. In order to give an adequate semantics to SystemC without inherent limitations on the subset supported, we have no choice than to formalize the execution semantics of the scheduler : we have either to include the scheduler semantics in our formalization or to use an existing formal model to represent separately the scheduler. Besides, the target formalism (1) should be expressive enough to allow the encoding of the program's control flow and (2) should have connections with existing tools for programs validation, such as model-checkers. This latter point allows to make concrete experiments and to compare the given formal semantics to the actual behavior of SystemC. Figure 13.1 represents the general approach for such formalization. At the beginning of the

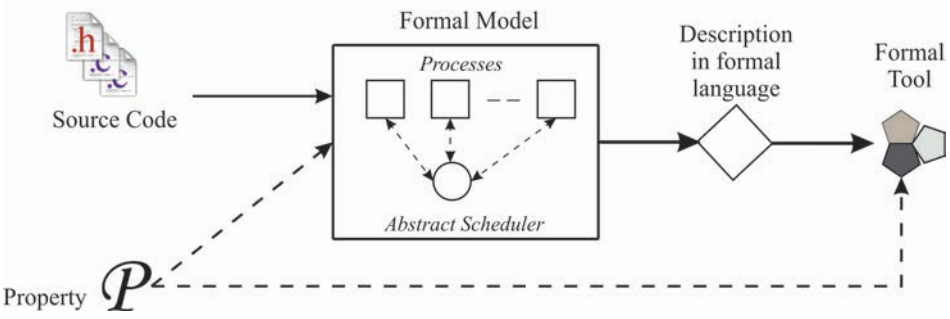


FIGURE 13.1 – *General Approach for Formalizing SystemC*

formalizing process, the source code of the application under study is analyzed. The goal of the analysis is to extract a formal model of the program, expressed in a chosen formalism.

This model contains separate unit of concurrency, that we call processes in the figure, and that have been identified previously in the source code. These processes interact with an abstract scheduler which is a model of the real scheduler used to execute and manage the various processes. This abstract scheduler is not extracted from source code but rather integrated in the abstract semantics of the language under study. Once the execution semantics of the scheduler are formally defined and included in the abstract representation of the processes, it may be translated to various forms, but ultimately, it has to be defined in a format that can be exploited by a formal tool. We call this format the formal language on the figure ; it covers both full-fledged formal programming languages such as Lustre [JLBP85] or *specification languages* like PROMELA [spi]. Separately, the property to be verified on the program can sometimes be integrated within the abstract model of processes or it can be given directly to the formal tool. This latter is generally a model-checker, that will provide automatically the result, and a counter-example leading to the error, if the property does not hold. Some tool chains also provide simulators, which are useful to check that the formal description does indeed express what was intended.

13.2 Formal Semantics of SystemC

We use a small-step semantics to define the formal semantics of a subset of SystemC language. Our semantics are based on the work of Shyamasundar, et al. [RSK07]. But our semantics are expressed in a different way in order to : first respect the syntax of the SystemC waiting-state automta (as defined in Chapter 7) and second express differently the simulation semantics of the SystemC scheduler.

In our semantics framework, we add request updates for channels which was not treated in [RSK07] in addition to the use of environment events and to global variables. Different changes generated during the execution of the program are expressed on transitions like in [RSK07]. We consider channels updates because they represent a mandatory step in the simulation semantics of the scheduler. The update phase is the step that comes immediately after the evaluation phase, where variables and channels are updated. In addition to request updates of channels, we consider also notification of immediate, delta and timed events which is conform to the semantics of the SystemC waiting-state automata. Besides, in our transitions, we express the output environment in the lower part while the input environment is expressed in

the upper part unlike in [RSK07]. We use this notation in order to respect the formalism of the SystemC waiting-state automata : the abstract formal representation that we use throughout this thesis to model SystemC designs.

We use the notation of the Structural Operational Semantics (SOS) of Plotkin [Plo04] to define a small step semantics of a subset of SystemC language. Those semantics compositionally capture all possible behaviors computed by a SystemC program. We suppose that each module is behaving either locally using its local variables or is communicating with the environment through the environment variables. The *local* variables are output signals, internal variables, output channels, output events, and the program counter for the process. The *environment* variables are input signals, input events, input channels, and global variables. As regards to the execution semantics of SystemC scheduler [WRM01], there is at most one process that is reacting to the environment. We can locally visualize instants during which reactions occur by observing the state (C++ variables and program counters for each processes) of the program, denoted σ, or the modeling environment (events, channels, signals, processes, etc), denoted E.

To describe how a statement changes the configurations of the environment, we write the transitions rules for processes as mentioned below :

$$\langle stmt, \sigma \rangle \xrightarrow[E_o]{E} \langle stmt', \sigma' \rangle$$

where :

- *stmt* is a SystemC statement that corresponds to the location of the program counter, before the reaction, and *stmt'* is the statement with the location of the program counter after the transition,

- σ and σ' are the states before and after the reaction respectively. They represent a function : $\mathcal{V} \cup \mathcal{CH} \mapsto values$, where $\mathcal{V}$ is the set of local and shared variables and $\mathcal{CH}$ is the set of channels.

- E is the environment (set of events and variables that activate the process) while the transition is executed, E_o is the output emitted during the transition. In general, an environment is a 5-tuple $E = (E^I, E^\delta, E^T, \mathcal{V}, \mathcal{RQ})$ where :

- E^I is the set of immediate events,
- E^δ is the set of next delta events,
- E^T is the set of timed events,
- $\mathcal{V}$ is the set of next delta updates for variable.
- $\mathcal{RQ}$ is a sequence consisting of pending requests to update channels. A request is a pair $(ch, exp(\sigma))$ where $ch \in \mathcal{CH}$ and $exp(\sigma)$ represents the value assigned to ch.

To denote that the output environment E_o remains unchanged we use the symbol $-$, i.e there is no events emitted during the transition, variables remain unchanged and channels are not modified. Our semantics are similar to the semantics of Shyamasundar [RSK07] where a complete behavioral semantics of SystemC is proposed. Here, we stress specifically on three main points :

- to capture all reactive features of SystemC.
- to specify a network of synchronous and asynchronous components computing either high-level transactions or low-level event communications.
- to specify two time scales : the delta cycle and the simulation time.

Besides, during our formalization and especially during parallel composition, we distinguish between the three phases of the simulation process (as response to the SystemC Scheduling algorithm (as mentioned in Figure 17.1) : this is the main contribution of our semantics. We `hide` the scheduler in a special parallel composition of concurrent processes, this composition is independent from the scheduler itself. The scheduler is then abstracted from the modeling process but already mainly presented within the parallel composition.

Our semantics have two main benefits : First, they start from a low-level description of SystemC (the delta-cycle) which puts in evidence the scalability of the global approach. Second, we don't need to model the scheduler using our formal model, we just build the automaton for each process. Thus the composed automaton will be independent from the scheduling policy and we gain in terms of modeling cost and verification cost. In this section we will present semantics for some sequential constructs (including assignments, channel statements, event statements, guarded statements and wait statements) and parallel composition where we distinguish between the three steps of SystemC semantics which is the main contribution

assignment	$< var\ v, \sigma > \xrightarrow[\_]{\_} < \epsilon, \sigma[v] >$
if	$\frac{<b,\sigma> \rightarrow <true,\sigma> \quad <p,\sigma> \rightarrow <\epsilon,\sigma'>}{<if\ b\ then\ p\ else\ q,\sigma> \xrightarrow[\_]{\_} <\epsilon,\sigma'>}$ $\frac{<b,\sigma> \rightarrow <false,\sigma> \quad <q,\sigma> \rightarrow <\epsilon,\sigma'>}{<if\ b\ then\ p\ else\ q,\sigma> \xrightarrow[\_]{\_} <\epsilon,\sigma'>}$
while	$\frac{<b,\sigma> \rightarrow <true,\sigma> \quad <p;\ while\ (b)\ do\ p,\sigma> \rightarrow <\epsilon,\sigma'>}{<while\ (b)\ do\ p,\sigma> \xrightarrow[\_]{\_} <\epsilon,\sigma'>}$ $\frac{<b,\sigma> \rightarrow <false,\sigma>}{<while\ (b)\ do\ p,\sigma> \xrightarrow[\_]{\_} <\epsilon,\sigma>}$

TABLE 13.1 – The structural operational semantics of SystemC statements

of our work compared to existing works on formalizing SystemC semantics using the SOS notations.

13.2.1 Basic Statements

The execution of an assignment $v := exp$ is instantaneous. It assigns the value of the expression *exp* to the variable v and keeps unchangeable the other variables and channels.

$$\frac{v \in \mathcal{V}}{\langle v := exp, \sigma \rangle \xrightarrow[\_]{E} \langle \epsilon, \sigma[v/exp] \rangle}$$

Transition rules for the statements : *skip*, *var v* as well as conditions and iterations are defined in the table below. These rules are similar to the rules presented in [Zhu05].

13.2.2 Channel Statements

The execution of channel statements involves two cases :

- Executing an output statement that we note $ch!!exp$ generates a request to update the channel *ch* with the expression *exp* and leaves other variables and channels unchanged. All pending requests are carried out in the following update phase. The newly generated request will remove the existing one from the request queue. We use operator \ to represent removing all elements from the request queue. (1) shows the transition rule of the channel output statement.

- Execution an input statement that we note $ch??v$ assigns the current value of channel

ch to the variable v and leaves other variables and channels unchanged. (2) shows the transition rule of the channel input statement.

$$(1)\quad \frac{ch \in Channels \wedge \sigma(ch) \neq exp(\sigma)}{\langle ch!!exp, \sigma\rangle \xrightarrow[E^I,E^\delta,E^T,\mathcal{V},\mathcal{RQ}\setminus(ch,exp(\sigma))]{E^I,E^\delta,E^T,\mathcal{V},\mathcal{RQ}} \langle \epsilon, \sigma[v/exp]\rangle}$$

$$(2)\quad \frac{ch \in Channels, v \in \mathcal{V}}{\langle ch??exp, \sigma\rangle \xrightarrow[-]{E} \langle \epsilon, \sigma[v/ch]\rangle}$$

13.2.3 Event Statements

The event notification statement immediately emits an event e in the next environment, and terminates. The processes waiting on these events will unblock in either the synchronization with the next environment and the synchronization with the next delta environment respectively. According to the way an event is notified, there are three kinds of event notifications : immediate notifications $notify()$, delta notifications $notify\_\delta()$ and timed notifications $notify\_t()$.

The execution of *notify()* triggers event e immediately, which will activate all processes that are waiting for it. The immediate event notification also overrides the delayed notifications on the same event if it will be notified in later delta cycles. The execution of $notify\_\delta()$ is instantaneous, which results in some changes in E^δ and E^T , not only adding a delayed notification to some sets, but also overriding a delayed notification. A notification scheduled to occur earlier will always override the one scheduled to occur later. Transition rules for events notifications are as below :

$$\langle e.notify(), \sigma\rangle \xrightarrow[e,e,\emptyset,\emptyset,\emptyset]{E} \langle \epsilon, \sigma\rangle$$

$$\langle e.notify\_\delta(), \sigma\rangle \xrightarrow[\emptyset,e,e,\emptyset,\emptyset]{E} \langle \epsilon, \sigma\rangle$$

$$\langle e.notify\_t(), \sigma\rangle \xrightarrow[\emptyset,\emptyset,e,\emptyset,\emptyset]{E} \langle \epsilon, \sigma\rangle$$

13.2.4 Wait Statements

The behavior of the *wait* statement is to wait for an event e to be in the environment or for a timeout. It works as a synchronization between parallel processes. Syntactically, rules for *wait* statement must be defined as follows :

- Rule 1 defines that if an event e is not in the environment, the process continues to wait without doing anything.

- Rule 2 defines that if an event e is present in the environment, the *wait*statement terminates and reduces to nothing.

$$\frac{e \notin E}{\langle wait(e), \sigma \rangle \xrightarrow[-]{E} \langle wait(e), \sigma \rangle}$$

$$\frac{e \in E}{\langle wait(e), \sigma \rangle \xrightarrow[-]{E} \langle \epsilon, \sigma \rangle}$$

13.2.5 Function Calls

Each function f is called by one or more processes. In order to response to all functions calls, we need to `duplicate` and `rename` functions which are called multiple times, we ignore recursive calls. Function parameters and return values are not represented but can be taken into account by using global variables for both.

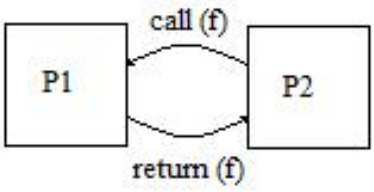


FIGURE 13.2 – *Function calls*

For each function f, we use a global Boolean variable F. We suppose that the effects of f are transformed as below :

- Calling f $\Leftrightarrow$ F=1

- Returning from the call $\Leftrightarrow$ {F==0}

- Begin of the declaration of f $\Leftrightarrow$ {F==1}

- End of the declaration $\Leftrightarrow$ F=0

The semantics of functions calls is defined as below : First a process P1 call the function f declared in a process P2 by affecting value 1 to the global variable F as mentioned previously.

Then, P1 **waits** for a response from the latter ; i.e, it tests if the value of F is equal to 0. The rule (1) presents the call step and rule (2) is for the return from call.

$$(1)\ \langle call(f), \sigma\rangle \xrightarrow[F=1]{E} \langle wait(), \sigma\rangle$$

$$(2)\ \langle f.return(), \sigma\rangle \xrightarrow[.]{E,F=0} \langle wait(), \sigma\rangle$$

13.2.6 Sequential Composition

If we consider two sequential processes P_1 and P_2, there are two cases for sequential composition : (i) If process P_1 does not terminate in the current instant, then P_2 cannot start. (ii) If P_1 terminates then P_2 starts in the environment in which P_1 terminates. Transitions rules are defined as follows :

$$\frac{\langle P_1, \sigma\rangle \xrightarrow[(E_1,E_1^{\delta},E_1^{T},\mathcal{V}_1,\mathcal{RQ}_1)]{E} \langle P_1', \sigma'\rangle}{\langle P_1; P_2, \sigma\rangle \xrightarrow[(E_1,E_1^{\delta},E_1^{T},\mathcal{V}_1,\mathcal{RQ}_1)]{E} \langle P_1'; P_2, \sigma'\rangle}$$

$$\frac{\langle P_1, \sigma\rangle \xrightarrow[(E_1,E_1^{\delta},E_1^{T},\mathcal{V}_1,\mathcal{RQ}_1)]{E} \langle \epsilon, \sigma'\rangle}{\langle P_1; P_2, \sigma\rangle \xrightarrow[(E_1,E_1^{\delta},E_1^{T},\mathcal{V}_1,\mathcal{RQ}_1)]{E} \langle P_2, \sigma'\rangle}$$

13.2.7 Parallel Composition

In this section, we consider transition rules for parallel composition. There are two kinds of configurations for parallel processes, one representing executing processes (processes that have been selected by the scheduler) and the other one representing processes that are not executing (either in a state waiting for an event or in a runnable state). In this section, we distinguish between the three phases in the simulation process of the SystemC scheduler (Figure 17.1), which is the main contribution of our semantics compared to those of [RSK07].

The Evaluation Phase

The evaluation phase starts from a non-empty table of runnable processes (i.e, processes that are waiting to be selected by the scheduler). Here we have two scenarios :

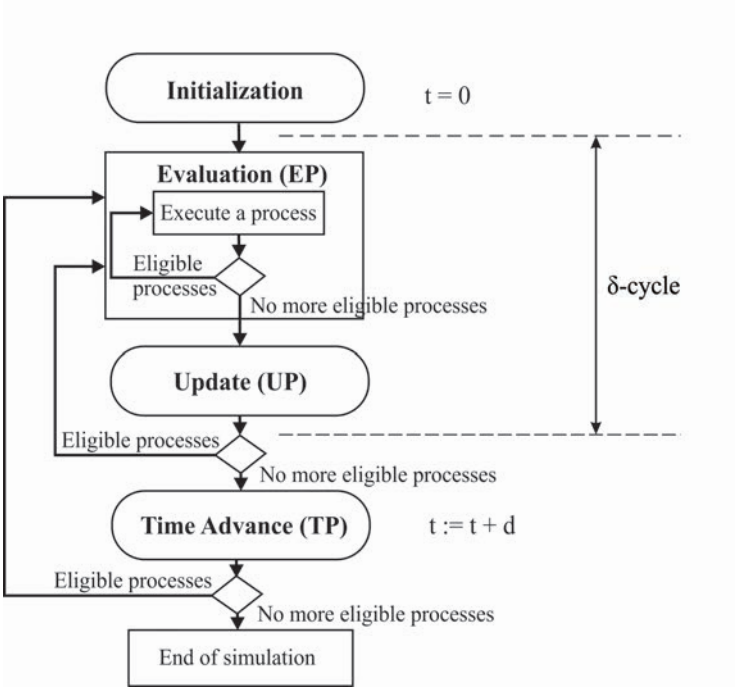


FIGURE 13.3 – *SystemC Scheduling Algorithm*

1. Immediate notifications of a set of events with processes waiting for them.

2. No immediate notifications but there is a non-empty set of runnable processes.

Transition rules for the evaluation phase are presented in Table 13.2.

1. Rule (1) : The immediate composition is defined to unblock all processes that are waiting for events present in the environment. We use the expression *waiting(P,e)* to denote that the process P is waiting for the event e. In other words, we may write the process P in a sequential form *wait ;P'*. It is a synchronous composition, but only for the wait statements.

2. Rule (2) : when all current processes are hung up *waiting* for events. The scheduler selects a process from the set of *ready* processes. This process runs until it reaches the next *wait* state. The function *add* guarantees that next-delta events and next-delta modifications of variables is taken into account in the process of scheduling the concurrent processes so that non deterministic behavior can be detected, i.e, non-deterministic behavior is possible when two or more different values can be written to a signal at the same evaluation phase, it must guarantees that the assigned value to the signal is exactly the

last value taken by the signal.

Update Phase

If there is no runnable processes, the scheduler updates channels (ch) with new data values (v). Channel update is carried out in the way of FIFO (First In First Out). During this phase some events are notified and those events may activate processes waiting for them. Rules are defined in Table 13.2.

Delta-cycle Advancing Phase

We will define rules for synchronization on delta events to build the next micro-environment. The rule proceeds only when there is no immediate events and there exists some delta events. The transition makes the delta events in E^{δ} become the immediate events in the next instant, and updates the state of variables. Here, we resume rules for the evaluation and the update phases since we are dealing with a new delta cycle. Rules for the delta cycle advancing phase phase are defined in Table 13.2.

Simulation Time Advancing Phase

Here, we define rules for the synchronization on timed events which builds the next environment from time events and advance macro-time (time simulation). It is effective when all processes are blocked, where there are no immediate events nor delta events. Timed events are posted by wait(time) statements, timers and clocks. We define here the same rules for the evaluation and the update phases. Rules for the delta cycle advancing phase phase are defined in Table 13.2.

13.3 Conclusion

In this chapter, we presented a small-step semantics of a subset of SystemC language. This subset includes the basic constructs of the language, communications using events and channels, parallel communication between synchronous/asynchronous processes and simulation at both the delta-cycle and transaction levels. The previous study is sufficient for our analysis in this thesis but it can be extended for more constructs. As we previously mentioned, the framework is based on the work of [RSK07]. But our semantics are expressed differently in order to be first conform to the formalism of the SystemC waiting-state automta and second in order to express differently the simulation semantics of SystemC scheduler.

In our semantics framework, we consider request updates for channels which was not treated in [RSK07] in addition to environment events and to global variables. All these information are expressed on transitions like in [RSK07]. In fact, we consider channels updates because they represent an important step in the simulation process of the scheduler. The update phase is the step that comes immediately after the evaluation phase, where variables and channels are updated with the final modification of global variables. In addition to request updates of channels, we consider also notification of immediate, delta and timed events which is conform to the semantics of the SystemC waiting-state automata. Besides, in transitions, we express the output environment in the lower part while the input environment is expressed in the upper part unlike in [RSK07]. This notation is also conform to the semantics of the SystemC waiting-state automata.

To study the parallel composition of concurrent processes, we choose to not model the scheduler separately, but to include it in the semantics of the parallel composition of the processes. In fact, we distinguish between different steps of the SystemC simulation semantics. Although, in work of [RSK07], authors choose to model separately the scheduler which requires to fix in advance the simulation policy, which is avoided in our semantics. Including SystemC simulation semantics in the formal semantics of the program, help us to express more details about the system parallel behavior independently from the simulation policy or without having to model separately the scheduler.

Our semantics are complete enough to handle all the simulation semantics of the SystemC language and to express in details the concurrent behavior of parallel processes that are locally independent but are communicating between each other through the environment

variables. Those semantics can be extended to handle more structures like information about the continuous time.

(Evaluation Phase)

(1)

$$\frac{\forall i{\in}\{1..n\},\exists e{\in}E^I,waiting(P_i,e)\ \wedge\forall j{\in}\{n{+}1..m\},\forall e{\in}E^I,\neg waiting(P_j,e)}{\langle P_1\|\cdots\|P_n\|\cdots\|P_m,\sigma\rangle \xrightarrow[(\emptyset,E^\delta,E^T,V^\delta,\mathcal{RQ})]{E} \left\langle P_1'\|...P_n'\|\cdots\|P_m,\sigma'\right\rangle}$$

(2)

$$\forall i \in \{1\dots n\},\ waiting(P_i)\ \ \forall j \in \{n+1\dots m\}, ready(P_j)$$

$$select\ p \in \{n+1\dots m\}, \langle P_p,\sigma\rangle \xrightarrow[(E_p^I,E_p^\delta,E_p^T,V_p^\delta,\mathcal{RQ}_p)]{E} \left\langle P_p',\sigma'\right\rangle$$

$$\frac{add(\left\langle E_p^\delta,E^\delta\right\rangle,\left\langle E_p^T,E^T\right\rangle,\left\langle V_p^\delta,V^\delta\right\rangle)}{\langle P_1\|\cdots\|P_n\|\cdots\|P_p\|\cdots\|P_m,\sigma\rangle \xrightarrow[(E_p^I,E_p^\delta\cup E^\delta,E_p^T\cup E^T,V_p^\delta\cup V^\delta,\mathcal{RQ}_p\cup\mathcal{RQ})]{(E^I,E^\delta,E^T,V^\delta,\mathcal{RQ})} \left\langle P_1'\|\cdots\|P_n'\|\cdots\|P_p'\|\cdots\|P_m,\sigma'\right\rangle}$$

(Update Phase)

$$\frac{\forall (ch,v){\in}RQ}{\langle P_1\|\cdots\|P_n,\sigma\rangle \xrightarrow[(E^I,E^\delta,E^T,V^\delta,\emptyset)]{(E^I,E^\delta,E^T,V^\delta,\mathcal{RQ})} \langle P_1\|\cdots\|P_n,\sigma[v/ch]\rangle}$$

(Delta Advancing Phase)

$$\frac{\forall i{\in}\{1..n\},waiting(P_i)}{\langle P_1\|\cdots\|P_n,\sigma\rangle \xrightarrow[(E^\delta,\emptyset,E^T,\emptyset,\mathcal{RQ})]{(\emptyset,E^\delta,E^T,V^\delta,\mathcal{RQ})} \left\langle P_1\|\cdots\|P_n,\sigma[V^\delta/V]\right\rangle}$$

(Timed Advancing Phase)

$$\frac{\forall i{\in}\{1..n\},waiting(P_i)}{\langle P_1\|\cdots\|P_n,\sigma\rangle \xrightarrow[(E^T,\emptyset,\emptyset,\emptyset,\mathcal{RQ})]{(\emptyset,\emptyset,E^T,\mathcal{RQ})} \left\langle P_1\|\cdots\|P_n,\sigma[V^\delta/V]\right\rangle}$$

TABLE 13.2 – Semantics for Parallel Composition

CHAPTER 14

Applying Static Analysis to Automatically Generate the SystemC Waiting State Automata

Static analysis, consists in analyzing the program by examining its source code without actually executing it on concrete inputs. A common paradigm in static analysis, which is also used in program verification, is symbolic execution [Kin76, Dar88] : the analyzed program is *executed*, but with symbolic instead of concrete values for the program variables. Symbolic execution is the most effective method to generate the set of all possible traces by symbolically executing the program. It still be not effective, but as precise as possible. Since the number of symbolic states may be infinite, we resort to abstraction techniques for computing and storing abstract states during symbolic execution. This enables analysis of an *under-approximation* of the program behavior. We resort to abstraction techniques and more precisely to predicate abstraction [GS97] to :

1. build the set of transitions from one wait state to another by merging the set of traces.

2. reduce unfaisable transitions from the set of transitions generated in (1).

Thus, the problem of building the SystemC waiting-state automata is reduced to the simpler problem of guessing potentially useful predicates.

14.1 Abstracting Low Level Semantics of SystemC to High Level Semantics

14.1.1 Introduction

A SystemC program is composed of a set of processes (or threads), each process is either operating locally or communicating with other processes using the wait statements. The idea as we previously presented (see Chapter 7) is to build the automaton for each process independently and then compose all automata together. But, Before composing the automata, we have first to symbolically execute the SystemC code.

Why we need to symbolically execute the program ? First, because symbolic execution uses symbolic values, instead of actual data to represent the values of program variables as well as the input values. As a result, the output values computed by a program are expressed as a function of symbolic values. Second, symbolic execution helps to capture conditions and

actions over transitions and it accumulates the environment effects over system variables and events.

14.1.2 Example

As an example of SystemC threads, we take the example of the *Timer* program expressed in SystemC. The structure of the thread is shown above : The process is sensitive to the *clock* and to the *start* signal. When start is active, the counter is decremented until it reaches 0, therefore the signal *timeout* is set to *true*. We take such an example because it is the adequate one to show the different transformations we made during symbolic execution of the program. In this example, we have different SystemC constructs : iteration, condition, assignment and the wait statement.

```
1 #include "systemc.h"
2 SC_MODULE (Timer){
3 ...
4 void timer(){
5 while(true){
6 if(start){
7  count=5;
8  timeout = false;
9  start = false;
10     }else if ( count > 0 ){
11       count--;
12       timeout = false;
13     }  else{
14           timeout = true;
15            }
16 }
17 wait();
18 }
19 SC_CTOR(Timer){
20 SC_METHOD (timer);
21 sensitive << clock << start;
22 }
23 };
```

The program is first visualized as a *control flow graph* (CFG). The nodes of this graph represent the basic commands and guard expressions of the thread, and the edges stand for flow of control between the nodes. We annotate the (CFG) with exemplary of logical expressions defined over variables, that we call the *path condition* (PC). We explain it later with more details.

14.1.3 Extended Symbolic Execution

The symbolic execution (SE) as first introduced by [Kin76, Dar88] is a natural extension of normal execution providing normal computation as a special case. The main idea behind SE is the use of symbolic values instead of the real ones in order to generate the set of all possible executions for all the values of the input variables. The semantics and rules of the symbolically executed program remain the same and need just to be extended in order to deal with the symbolic values. Therefore, the assignment operation is fairly clear, the assigned variable changes its interpretation by evaluating the expression to the right that consists in replacing all the symbolic values that it contains with their corresponding symbolic expressions. As for the conditional instructions a choice has to be made in order to decide what branch should be taken. If we apply symbolic execution to the *Timer* example, then we generate the graph in Figure 14.1.

Throughout this thesis, the program is then first visualized as a *control flow graph* (CFG) (Figure 14.1). The nodes of this graph represent the basic commands and guard expressions of the thread, and the edges stand for flow of control between the nodes. We annotate the (CFG) with exemplary of logical expressions defined over variables : the assignment statement is transformed into equality written between accolades and the *path condition* (PC) that we define over conditional instructions. The *PC* is a (*quantifier-free*) boolean formula defined over the symbolic inputs, it accumulates constraints which the inputs must satisfy so that the execution follows the particular associated execution path. We suppose here that the *PC* is also a first-order formulas which always hold when control flow reaches a specific program point such as a loop entry. Therefore a path condition (PC) is also included in the state that will keep track of all the decisions made along the execution, working as an accumulation of assertions made on that symbolic variables, refining their values domains and helping decide which of the **then** or the **else** branches should be taken. We can see that, by construction,

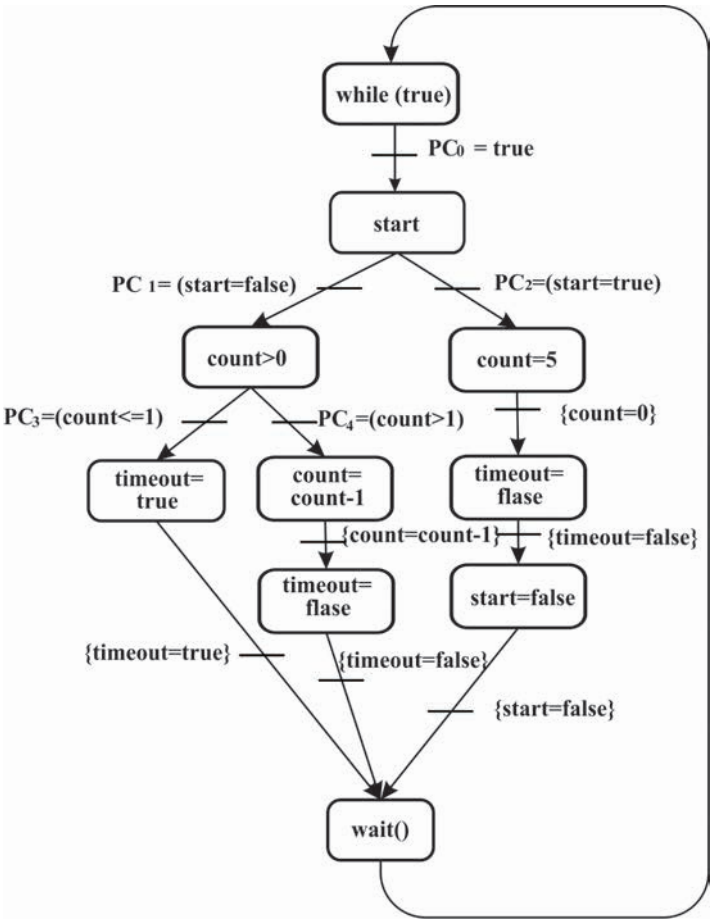


FIGURE 14.1 – *The control flow graph of the Timer process*

the SE only generates feasible paths.

The state of a basic SE is composed of the values of the current variables in use and the path condition (PC) that represents the history of the choices made up to that point, mostly present to deal with the conditional instructions. Besides, the state is composed of the emphinput events that triggers the transition, the *output* events triggered during the transition and a function f that modifies the variables. Formally, we write : $S = (e_{in}, PC, e_{out}, f)$ where :

- e_{in} is the set of events that activates the state.
- PC is the path condition.
- e_{out} is the set of events triggered.
- f is the effect function.

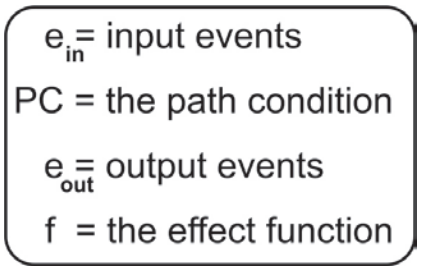


FIGURE 14.2 – *The Symbolic State generated during Symbolic Execution*

14.1.4 Abstracting Operational Semantics of a Subset of SystemC

In this section, we modify rules in Section 13 to suit the formalism of the System waiting-state automata [HM12] : an abstract formal model used to model/verify system designs written in the SystemC language.

A SystemC waiting-state automata A is defined as a transition system over a set $\mathcal{V}$ of variables. It is a tuple $A = (S; E; \mathcal{T})$, where S is a finite set of states , E is a finite set of the environment events and $\mathcal{T}$ is a finite set of transitions where every transition is a 6-tuple $(s; e_{in}; p; e_{out}; f; s')$:

- s and s' are two states in S, representing respectively the initial state and the final state ;
- e_{in} and e_{out} are two sets of events : $e_{in} \subseteq E; e_{out} \subseteq E$;
- p is a predicate defined over variables in $\mathcal{V}$, i.e., $FV(p) \subseteq \mathcal{V}$, where *FV(p)* denotes the set of free variables in the predicate p ;
- f is an effect function over $\mathcal{V}$;

We often write $s \xrightarrow[e_{out},f]{e_{in},p} s'$ for the transition $(s; e_{in}; p; e_{out}; f; s')$. The triggering of such transition for a process P is captured by the following intuitive characterization :

If

(i) The process P is invoked in an initial state that satisfies p, and

(ii) at any moment during the computation of P the event e_{in} exists in the input environment,

then

(iii) the event e_{out} is triggered and we add it to the output environment,

(iv) if this computation terminates, the final state executes f.

Therefore, during SE, we define our execution traces as follows :

$$\langle stmt_0, \sigma_0 \rangle \xrightarrow[e^1_{out}, f^1]{e^1_{in}, p^1} \dots \xrightarrow[e^n_{out}, f^n]{e^n_{in}, p^n} \langle stmt_n, \sigma_n \rangle$$

Where $stmt_0$ and $stmt_n$ present the first and the final statements, $\{e^1_{in} \dots e^n_{in}\}$ is the set of input events of the transitions, $\{e^1_{out} \dots e^n_{out}\}$ is the set of the output events, $\{p^1 \dots p^n\}$ is the set of predicates and $\{f^1 \dots f^n\}$ is the set of functions. The set of path conditions determine the set of predicates mentioned in the expression above and the set of assignment statements is used to determine the functions. We use the input and the output environment E and E_o to determine the input events e^i_{in} and the output events e^i_{out}, $i \in [1..n]$. During the execution of the process from $stmt_0$ to $stmt_n$ (or between σ_0 and σ_n), the states are observable only from within the process, and no other process in the environment can observe the intermediate states. Hence, from the environment point of view, only the first and the last states are observable. In Table 14.1, we represent new rules for the *wait* and the *notify* statements, along with the three basic programming constructs in imperative languages : *assignments*, *conditional statements*, and *loops*. In fact each transition has : (i) a set of pre-conditions : an input event e_{in} that triggers the transition and a predicate p over variables and (ii) a set of post-conditions : an output event e_{out} and a function f over predicates.

We resume the example of the Timer that we symbolically execute in order to generate its CFG. The symbolic execution of the program as shown in Figure 14.1, begins with the first statement : *while(true)*, we have to deal with two cases : the loop is not entered and the loop is unfolded at least once. Technically, we apply the *while* rule (Table 14.1), in this case we don't have two branches since the guard is usually true. The next step in the symbolic execution is to deal with the *if* statement $if(start)$, which produces two conjuncts in place of one. The first conjunct yields to a set of assignment statements {count=5, timeout=false, start=false} : this conjunct initializes the set of input variables, we apply the assignment rule (Table 14.1). The effect of the assignment manifests itself in the set of following assumptions $(count = 5, timeout = false, start = false)$.

The second conjunct yields to the second *if* statement $if(count > 0)$ which yields to two additional branches so two additional traces : this branch constitutes two traces from the first state to the first waiting state : the *wait* statement. The transition rule for the wait statement

assignment	$< x := e, \sigma > \xrightarrow[\{\},x=e]{\{\},()} < \epsilon, \sigma[x/e] >$
if	$\dfrac{<b,\sigma>\rightarrow<true,\sigma> \quad <p,\sigma>\rightarrow<\epsilon,\sigma'>}{<if\ b\ then\ p\ else\ q,\sigma> \xrightarrow[\{\},id]{\{\},(b=true)} <\epsilon,\sigma'>}$ $\dfrac{<b,\sigma>\rightarrow<false,\sigma> \quad <q,\sigma>\rightarrow<\epsilon,\sigma'>}{<if\ b\ then\ p\ else\ q,\sigma> \xrightarrow[\{\},id]{\{\},(b=false)} <\epsilon,\sigma'>}$
while	$\dfrac{<b,\sigma>\rightarrow<true,\sigma> \quad <p;\ while\ (b)\ do\ p,\sigma>\rightarrow<\epsilon,\sigma'>}{<while\ (b)\ do\ p,\sigma> \xrightarrow[\{\},id]{\{\},(b=true)} <\epsilon,\sigma'>}$ $\dfrac{<b,\sigma>\rightarrow<false,\sigma>}{<while\ (b)\ do\ p,\sigma> \xrightarrow[\{\},id]{\{\},(b=false)} <\epsilon,\sigma>}$
wait(e)	$\dfrac{e\in E}{<wait(e),\sigma> \xrightarrow[\{\},id]{\{e\},()} <\epsilon,\sigma>}$ $\dfrac{e\notin E}{<wait(e),\sigma> \xrightarrow[\{\},id]{\{\},()} <wait(e),\sigma>}$
e.notify()	$\dfrac{add(e,E)}{<e.notify(),\sigma> \xrightarrow[\{e\},id]{\{\},()} <\epsilon,\sigma>}$

TABLE 14.1 – The abstracted operational semantics of SystemC statements

is the same as in Table 14.1 with only one entry condition which is the clock (clk) ($e_{in} = clk$) and no exit conditions : this is the last statement in the program. Since we have a loop, we turn back to the first state and then we build another trace from the waiting state to the first state.

Now, we have to extract the waiting state automaton from the CFG of the Timer process : as you notice in the control flow graph in Figure 14.1, we have just one waiting state, thus the automaton will have just one state and contains only loops. In the CFG, we have three traces from and to the waiting state, thus we have three loops in the automaton. Although, there is one loop that is triggered only one time. This loop represents the trace below :

$\langle if(start), \sigma\rangle \xrightarrow[\{\},id]{\{\},(start=true)} \langle true, \sigma\rangle ; \langle count = 5, \sigma\rangle \xrightarrow[\{\},count=5]{\{\},()}$
$\langle \epsilon, \sigma\rangle ; \langle timeout = false, \sigma\rangle \xrightarrow[\{\},timeout=false]{\{\},()} \langle \epsilon, \sigma\rangle ; \langle start = false, \sigma\rangle \xrightarrow[\{\},start=false]{\{\},()} \langle \epsilon, \sigma\rangle .$

The set of transitions that activate that trace will be triggered once, since in the last statement *start* is set to *false* which means that we will never enter that branch next time, which means that this trace will never be activated again. As a result, the waiting state automata of the Timer process will have just one state and two loops as shown in Figure 14.3.

We use the conjunction $\wedge$ to accumulate predicates and the order of predicates is preserved.

$$\langle if(start),\sigma\rangle \rightarrow \begin{cases} \frac{\{\},(start=true)}{\{\},id}; \frac{\{\},()}{\{\},count=5}; \frac{\{\},()}{\{\},timeout=false}; \frac{\{\},()}{\{\},start=false} \\ \frac{\{\},(start=false)}{\{\},id}; \begin{cases} \frac{\{\},(count<=0)}{\{\},id}; \frac{\{\},()}{\{\},timeout=true} \\ \frac{\{\},(count>0)}{\{\},id}; \frac{\{\},()}{\{\},dec}; \frac{\{\},()}{\{\},timeout=false} \end{cases} \end{cases}$$

$$\langle if(start),\sigma\rangle \xrightarrow[\{\},id]{\{\},(start=true)} \langle true,\sigma\rangle ; \langle count=5,\sigma\rangle \xrightarrow[\{\},count=5]{\{\},()} ..$$

$$\langle if(start),\sigma\rangle \xrightarrow[\{\},id]{\{\},(start=false)} \langle false,\sigma\rangle ; \langle if(count>0),\sigma\rangle \rightarrow ..$$

$$\langle if(count>0),\sigma\rangle \rightarrow \begin{cases} \frac{\{\},(count>0)}{\{\},id}; \frac{\{\},()}{\{\},dec}; \frac{\{\},()}{\{\},timeout=false} \\ \frac{\{\},(count<=0)}{\{\},id}; \frac{\{\},()}{\{\},timeout=true} \end{cases}$$

$$\langle if(count>0),\sigma\rangle \xrightarrow[\{\},id]{\{\},(count>0)} \langle true,\sigma\rangle ; \langle count=5,\sigma\rangle \xrightarrow[\{\},count=5]{\{\},()} ..$$

$$\langle if(count>0),\sigma\rangle \xrightarrow[\{\},id]{\{\},(count<=0)} \langle false,\sigma\rangle ; \langle timeout=true,\sigma\rangle \xrightarrow[\{\},timeout=true]{\{\},()}$$

TABLE 14.2 – Operational semantics for the Timer

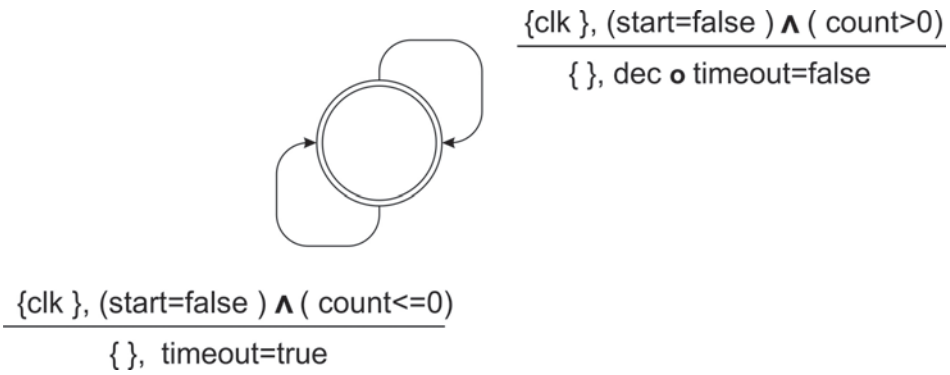


FIGURE 14.3 – *The WSA for the Timer*

We use also the symbol ∘ to compose functions.

14.1.5 The Abstracted Operational Semantics vs the SystemC Waiting State Automata Semantics

The abstracted operational semantics in Section 14.1.4 are generated from the semantics in Section 13. The latter was modified to suit the formalism of the waiting-state automata as presented in [HM12]. Changes concern especially entry and exit conditions on transitions. New semantics are accurate enough to give a low level representation of SystemC designs and generate all possible traces from the original code. In fact, those semantics describe different states of a process : (i) when it is locally behaving and (ii) when it communicates with its

environment. The main difference between the previous semantics and those of the SystemC waiting-state automata is that the semantics of the SystemC model is an abstraction or a high level representation of the operational semantics, where we consider only states when a process is visible by its environment. The advantage using the SystemC waiting-state semantics is that the local behavior of the process is no more represented when we model the whole system from the time it becomes not visible by the environment.

14.2 Predicate Abstraction

We resort to predicate abstraction as introduced first by [GS97] to (*i*) compute infer the relations between the transitions guards and affects and (*ii*) reduce the control flow graph (CFG) resulting from the symbolic execution by considering only special states that synchronize between the communicating processes (See Figure 14.4). The set of path conditions (PCs) we generate in Section 14.1.3 are ideal candidates for predicates, and thus, the path condition will definitely be a boolean combination of predicates. In this section, we illustrate the use of predicate abstraction on trivial examples, we define our *abstraction rule* to merge traces between each two waiting states. This rule is limited to programs without loops, we study a special case of a program with loop and we define a new methodology how to infer invariants for loops.

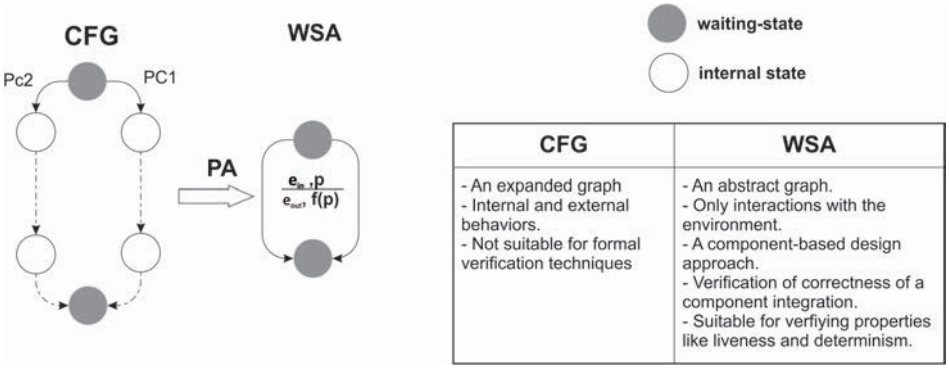


FIGURE 14.4 – *The use of PA to transform the CFG to the SystemC WSA*

14.2.1 Background

In predicate abstraction [TBR01, SS88], the concrete system is approximated by only keeping track of certain predicates over the concrete state variables. Each predicate corresponds

to an abstract boolean variable. Any concrete transition corresponds to a change of values for the set of predicates and it is subsequently translated into an abstract transition. Using this technique, it is possible to not only reduce the complexity of the system under verification, but also, for software systems, to extract finite models that are amenable to model checking algorithms. The technique of predicate abstraction was first used for verifying low level languages such as C. But the emergence of new languages describing systems on different levels of abstraction such as SystemC encourages researchers to apply this technique on them, accordingly several results were developed in this field [EC04].

Definition 1 Let $A = (S, T, I)$ be the state graph of a program P where S is the set of states, T is the set of transitions and I the set of initial states. Let $\widetilde{S}$ a lattice of abstract states and $(\alpha : P(S) \mapsto \widetilde{S}, \gamma : \widetilde{S} \mapsto P(S))$ a *Galois connection* [EC04] [1] , where the abstraction function α associates with any set of *concrete* states a corresponding *abstract* state (the abstract state space is a lattice where larger abstract states represent larger sets of concrete states). The concretization function γ associates with every abstract state the set of concrete states that it represents.

We assume that the abstract model can make a transition from a state $\widetilde{s}$ to a state $\widetilde{s}'$ iff there is a transition from s to s' in the concrete model, where $\widetilde{s}$ is the abstract state of s and $\widetilde{s}'$ is the abstract state of s' . We denote the transition relation :

$$R := \{(\widetilde{s}, \widetilde{s}') | \exists s, s' \in S : R(s, s') \wedge \alpha(s) = \widetilde{s} \wedge \alpha(s') = \widetilde{s}'\}$$

Definition 2 Formally, we assume that the program maintains a set of n predicates $\{p_1, .., p_n\}$ ordered by implication. These predicates are global, i.e., the abstract model only contains one set which is used by all the threads. A predicate p_i denotes the subset of states that satisfy the predicate $\{s \in S | s \models p_i\}$. The range of the abstraction consists of boolean formulas constructed using a set of boolean variables $\{B_1, .., B_n\}$ defined over the set of predicates and ordered by implication. When applying all predicates to a specific concrete state, we obtain a vector of n boolean values, which represents an abstract state $\widetilde{s}$. If X ranges over sets of concrete states and Y ranges over boolean formulas in $\{B_1, .., B_n\}$ then the abstraction and

1. a Galois connection is a pair of functions (α, γ) satisfying $\alpha(\gamma(\widetilde{s})) = \widetilde{s}$ and $\varphi \Rightarrow \gamma(\alpha(\varphi))$. Given γ, α is implicitly defined by $\alpha(\varphi) = \cap\{\widetilde{s} \in \widetilde{S} | \varphi \Rightarrow \gamma(\widetilde{s})\}$

the concretization function α and γ have the following properties :

$$\alpha(X) = \bigwedge\{Y | X \Rightarrow \gamma(Y)\}$$

$$\gamma(X) = \bigvee\{X | \alpha(X) \Rightarrow Y\}$$

The main challenge in predicate abstraction is to identify the predicates that are necessary for proving the given property. In work of [SO03], the predicate abstraction was applied to C programs, the authors have defined an algorithm for inferring predicates based on branch statements and using *weakest precondition* (WP). In this present work we suppose that the abstraction is applied on transitions instead of states like in work of [AP05].

Definition 3 A predicate transformer [Dij97] is a total function between two predicates on the state space of a statement. We distinguish two kinds of predicate transformers : the **weakest-precondition** and the **strongest-postcondition**. Technically, predicate transformer semantics perform a kind of symbolic execution of statements into predicates : execution runs backward in the case of weakest-preconditions, or runs forward in the case of strongest-post-conditions.

We focus here only on the weakest-precondition transformer : Given S a statement, the weakest-precondition of S is a function mapping any postcondition Q to a precondition. Actually, the result of this function, denoted $wp(S, Q)$, is the *weakest* precondition on the initial state ensuring that execution of S terminates in a final state satisfying Q. We show in Figure 14.5 the definition of weakest-precondition for some examples of sequential statements.

$$wp(skip, Q) \Leftrightarrow Q$$
$$wp(abort, Q) \Leftrightarrow true$$
$$wp(x := e, Q) \Leftrightarrow Q[x \leftarrow e]$$
$$wp(c_1; c_2, Q) \Leftrightarrow wp(c_1, wp(c_2, Q))$$
$$wp(if\ b\ then\ c_1\ else\ c_2, Q) \Leftrightarrow (b \Rightarrow wp(c_1, Q)) \wedge (\neg b \Rightarrow wp(c_2, Q))$$
$$wp(if\ b\ then\ c, Q) \Leftrightarrow (b \Rightarrow wp(c, Q)) \wedge (\neg b \Rightarrow Q)$$

FIGURE 14.5 – *Rules for weakest-precondition.*

Inferring Predicates Our predicates p are defined over the path conditions PC, they are first-order formulas defined as below :

$$p := n|x|p \odot p| \ominus p$$

Where :

- $\odot$: is a binary relation, it is one of the following : $+, *, -$ or any comparison : $\leq, \geq, <$ $, >, =, \neq$ or $\wedge, \vee$
- $\ominus$: is an unary relation (it can only be $\neg$)
- n : is an integer
- x : is a variable

To infer relation between predicates during traces fusion, we use the standard definitions for constructively computing **weakest precondition** (Figure14.5) : For all the pair of observable points (paths from a wait statement to the next wait statement in the graph) we compute the weakest precondition between the waiting states, and add it to the waiting state automaton if the weakest precondition is satisfiable. Note that one cannot constructively compute the weakest precondition for loops that cannot be unrolled. This is why we present separately how to compute predicates for programs with loops, we illustrate this on an example.

Conclusion Our method works by analyzing the concrete model in order to make the state space small and finite. Therefore, we abstract the system along several orthogonal dimensions :

- ❄ Control Flow Abstraction : SystemC semantics usually contain *branch* statements and iterations such as the *if* statements. The *if* statement has two branches, we call the boolean predicates that determine which branch to be executed, branch conditions or path conditions. We intend to extract the branch conditions and use them as predicates in predicate abstraction. Otherwise, we abstract the control flow representation of the SystemC thread into an abstract model that (1) has fewer locations, and (2) over approximates the behavior of (e.g simulates) the thread it represents. At each control state the thread verifies a condition (c) on variables or signals and executes an action (a) over the same set. Predicates are then defined over c and a.

- Event Abstraction : Here we consider a special statement defined either over clock or channel events in SystemC : this is the famous *wait/notify* mechanism which characterizes SystemC semantics. The thread is either waiting for an activating (input) event e_{in} or for the notification of an (output) event e_{out}.

- Data Abstraction : Predicate abstraction is suitable for handling variables with large domains. Such variables are usually called data variables. For a set of predicates $P(V)$ and a formula φ over V (V the set of variables). Data abstraction consists yet in replacing important formulas over concrete data variables with abstract predicates, it is possible to significantly reduce the complexity of verification. This abstraction is in fact the basis of the first abstraction where conditions are defined over variables.

14.2.2 Handling Execution Traces Without Loops

We define each execution trace generated during symbolic execution as follows : it starts from a state that represents a *wait* statement and then we consider all the consecutive transitions that lead to the next *wait* statement in the control flow graph. Otherwise, σ_0 and σ_n represent wait states and all the intermediate states from σ_1 to σ_{n-1} represent the regular sequential constructs (including assignments, channel statements, event statements, guarded statements).

$$\langle stmt_0, \sigma_0 \rangle \xrightarrow[e^1_{out}, f^1]{e^1_{in}, p^1} \langle stmt_1, \sigma_1 \rangle \dots \xrightarrow[e^n_{out}, f^n]{e^n_{in}, p^n} \langle stmt_n, \sigma_n \rangle$$

The goal of this analysis is to explain how to generate one-transition system from a set of consecutive transitions (an execution trace). To do so, we define an abstraction rule that starts from an initial subset of predicates and defines different transformations applied to that subset. The purpose of this transformation is to build a candidate predicate for each transition in the SystemC waiting-state automata.

Algorithm

We consider P the set of predicates and F the set of functions. We use the standard definitions for constructively computing weakest precondition (wp) (Figure14.5). For all the pairs of wait states (paths from a wait statement to the next wait statement in the control flow graph (CFG)) we compute the weakest precondition between the points, and add it to

the SystemC waiting-state automaton if the weakest precondition is satisfied. But first, we consider the function F_E that describes the changes in the set of output events when we merge consecutive transitions. $F_{E_{out}}$ eliminates an output event e_{out} from the set of output events if it figures in the following input events E_{in} in the forward transitions, otherwise it adds e_{out} to the set of output events.

$$F_E(e_{out}, E^{out}) =_{df} \begin{cases} E^{out} \backslash \{e_{out}\} \ if \ e_{out} \in E^{in} \\ E^{out} \cup e_{out} \ otherwise \end{cases}$$

We formally define the following abstraction rule that transforms a series of transitions in an execution trace into a one-transition trace with only one entry state and one exit state :

$$\frac{\langle stmt_1, \sigma_1 \rangle \xrightarrow[e^*_{out}, f^*]{e^*_{in}, p^*} \langle stmt_n, \sigma_n \rangle}{\langle stmt_1, \sigma_1 \rangle \xrightarrow[e^1_{out}, f^1]{e^1_{in}, p^1} \dots \xrightarrow[e^n_{out}, f^n]{e^n_{in}, p^n} \langle stmt_n, \sigma_n \rangle} \tag{14.1}$$

where $e^*_{in} = \bigcup_{i=1}^{i=n} e^i_{in}$ and $e^*_{out} = \bigcup_i F_E(e^i_{out}, E^{out})$.

Now, we use predicate abstraction to infer the relation between the set of predicates p^i and the functions f^i in order to define how we generate p^* and f^* : For each predicate p^i, we select the subset of functions $F^i \subset F$ that modifies p^i in the transitions that are triggered before p^i. More precisely, any free variable of F^i is incorporated as terms of the predicate p^i, i.e, the predicate p^i is modified by F^i during the execution of the trace. The goal is then to compute for each p^i the set of weakest preconditions of the last function from the subset F^i with respect to p^i. Besides, we consider the same order of the functions f^i as in the initial execution trace, because in our study we will consider only the last function that modifies each predicate since the intermediate transitions are simultaneous.

In this analysis, we garantee that for each predicate p_i, it is possible to generate the associated weakest-precondition. First, because we have a finite set of distinct predicates defined on transitions in the SystemC waiting-state automata. Second, in our analysis, we consider only the affect of the last function that modifies each predicate, so we garantee that we can use the standard definition of the weakest-precondition defined in Subsection 14.2.1 to compute the weakest-precondition.

We call f^i_F : the last function in F^i that modifies p^i. We consider the following execution

trace where we consider only parameters predicate p and the effect function f :

$$\xrightarrow[f^1]{p^1}\xrightarrow[f^2]{p^2}\xrightarrow[f^3]{p^3}\dots\xrightarrow[f^{n-1}]{p^{n-1}}\xrightarrow[f^n]{p^n}$$

For each predicate p^i, each subset F^i of functions that modifies p^i and each function $f_F^i \in F^i$ that modifies the predicate p^i the last, we define the new set of predicates as follows :

$$\begin{cases} p'^1 = p^1 \\ p'^2 = \mathcal{WP}(f_F^2, p^2) \\ p'^3 = \mathcal{WP}(f_F^3, p^3) \\ .. \\ p'^n = \mathcal{WP}(f_F^n, p^n) \end{cases}$$

Where $\mathcal{WP}$ is the weakest precondition for the function f_F^i that verifies the predicate p^i. The previous formulas are valid only with one condition : when the free variables $\mathcal{FV}$ of the predicate p^i (where p^i is true in the present environment) are included in the modified variables $\mathcal{MV}$ of f_F^i (where f_F^i represents the previous environment in which the transition is taking place) ; i.e $\mathcal{FV}(p^i) \subset \mathcal{MV}(f_F^i)$. We use the conjunction $\wedge$ to accumulate predicates and the order of predicates is preserved. We use also the symbol $\circ$ to compose functions.
Then, we define the predicate p^* for the equation 1 as follows :

$$p^* = \bigwedge_i p'^i$$

f^* is the composition of all the functions f_F^i, i.e :

$$f^* = \underbrace{f_F^1 \circ .. \circ f_F^n}_{n\ times}.$$

We get as a result the configuration below which conforms to the SystemC waiting-state automaton definition :

$$\langle stmt_0, \sigma_0 \rangle \xrightarrow[e^*_{out}, f^*]{e^*_{in}, p^*} \langle stmt_n, \sigma_n \rangle$$

We do the same for all execution traces. The abstraction rule as defined in equation (13.1) is

only valid for *program without loops.*

Example

Consider the following program with two variables x and y, this program executes two tests on x and y and modifies both variables. We illustrate the previous results on this example.

```
1 if (x = 1){
2 x = x + 1;
3 y := y + 1;
4 } else {
5 if (y > 0) {
6 y := y - 1;
7 }}
```

We have two execution traces in this example :

$$\begin{cases} \xrightarrow{(x=1)} \xrightarrow[x=x+1]{} \xrightarrow[y=y+1]{} \\ \xrightarrow{(x\neq 1)} \xrightarrow{(y>0)} \xrightarrow[y=y-1]{} \end{cases}$$

For each trace, we determine the p^* then the f^*. But first, we fix the set of candidate predicates and functions. For lines 1, 4 and 5, we associate respectively the set of predicates p^1, p^4 and p^5. For lines 2, 3 and 6, we associate respectively the set of the following functions : f^2, f^3 and f^6. They are defined as follows :

$$\begin{cases} p^1 = (x = 1), p^4 = (x \neq 1) \ and \ p^5 = (y > 0) \\ f^2 = (x = x + 1), f^3 = (y = y + 1) \ and \ f^6 = (y = y - 1) \end{cases}$$

Let us consider the set of pairs (f^2, p^1) and (f^6, p^5). we define now : $p'^1 = \mathcal{WP}(f^2, p^1)$ and $p'^2 = \mathcal{WP}(f^6, p^5)$, the goal is to infer the new predicates from the initial set of predicates and consider functions (actions) that modify each predicate. This is the case of (p^1, f^2) and (p^5, f^6) since f^2 is the last and the only function that modifies p^1 and f^6 is the last and the

only function that modifies p^5. We get as a result two predicates :

$$\begin{cases} p'^1 = \mathcal{WP}(x = x + 1, (x = 1)) = (x = 2) \\ p'^2 = \mathcal{WP}(y = y - 1, (y > 0)) = (y \geq 0). \end{cases}$$

Now we determine the function f^* : let us consider first $F = \{x = x+1, y = y+1, y = y-1\}$, we consider the same order of the functions as in the execution trace and as we previously explain. We extract from F the subsets of functions that modify the same predicate from the initial set of predicates. Here, we have three subsets : $F^1 = \{x = x + 1\}$, $F^2 = \{y = y + 1\}$ and $F^3 = \{y = y - 1\}$. From F^1 we extract the function $f'^1 = f^2$ since we have just one element in this subset. From F^2 we extract the last and the only function that modifies the variable y, we get then the following function : $f'^2 = f^3$. As a result $f^* = f'^1 \circ f'^2 = f^2 \circ f^3$. Besides, we consider for the second execution trace the function $f'^* = f^6$.

To conclude, the first trace is transformed into one transition trace of the form : $s_0 \xrightarrow[f^*]{p^*} s_1$, such that $p^* = (x = 1)$ and $f^* = (x = x+1) \circ (y = y+1)$. The second trace is also transformed into the following transition : $s'_0 \xrightarrow[f'^*]{p'^*} s'_1$ such that $p'^* = p'^2 \wedge p^4$ and $f'^* = (y = y - 1)$.

As a result, we obtain the following transitions :

$$\begin{cases} s_0 \xrightarrow[(x=x+1)\circ(y=y+1)]{(x=1)} s_1 \\ s'_0 \xrightarrow[y=y-1]{(x\neq 1)\wedge(y\geq 0)} s'_1 \end{cases}$$

14.2.3 Handling Loops

As a simple example of SystemC threads, we consider the program above, that computes the maximal element of a table of positive integers T.

```
1 max=0;
2 i=0;
3 while(i < T.length){
4 if(T[i] > max) max= T[i];
5 i++;
6 }
```

Since loops constructs are of a notorious difficulty in the formal verification of programs, we will just focus on loops and how to use predicate abstraction to automatically infer invariants

for loops. Several attempts have been made to automatically infer invariants for loops using predicate abstraction, it was first introduced by [FQ02] and later used in [LL02]. Our method is based on predicate abstraction, an abstract interpretation technique [CC77] in which the abstract domain is constructed from a given set of predicates over program variables. A novel feature of our approach is that it infers predicates by iteration and in a simple way.

Throughout this paper, the program is first visualized as a *control flow graph* (CFG) (Figure 14.6). The goal of our approach is to prove that after executing the program above,

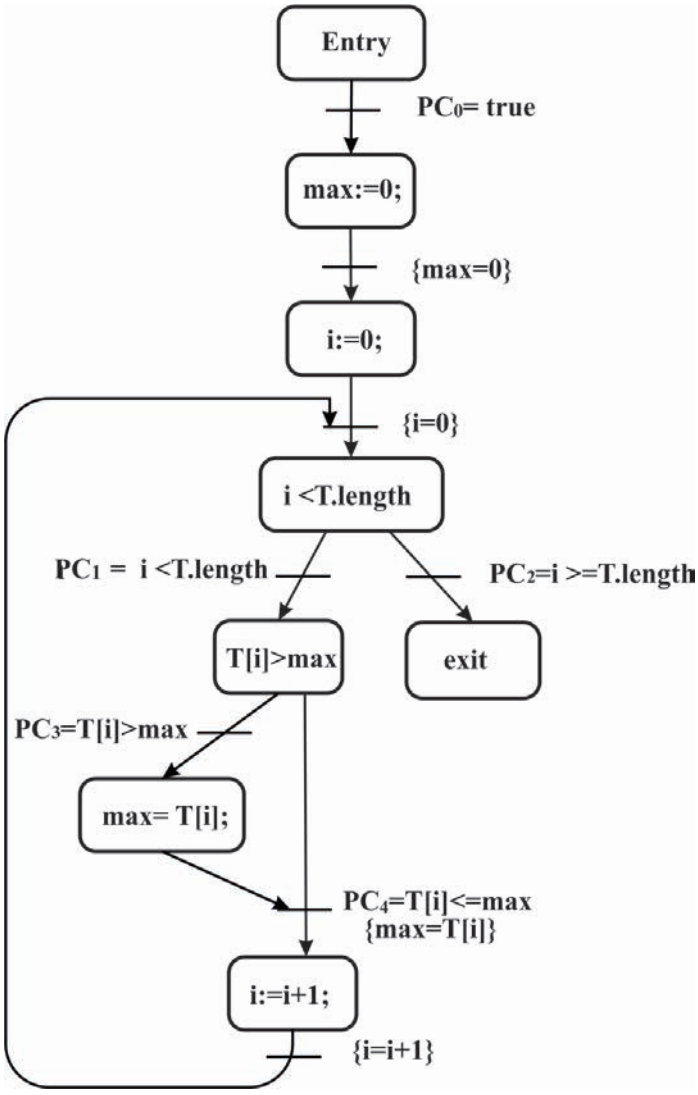


FIGURE 14.6 – *The Control Flow Graph generated during symbolic execution*

all elements of the array are less than or equal to **max**. The symbolic execution of the example is as follows : we execute first the assignment statements $inst.1 - 2$ (line 1 and 2) of

the program example. The effect of the two assignments manifests itself in the two additional assumptions $max = 0$ and $i = 0$. Now, the active statement of the program is the while loop. If we consider the formula we build before entering the loop as an invariant for the loop, we can then consider the formula φ_0 as a first candidate for the loop invariant. Our technique is similar to the work of [SEE07] where authors infer invariants for loops in Java programs, they use a special version of first-order predicate logic to express the semantics of Java. Their method is based on a combination of symbolic execution and computing fixed points via predicate abstraction. Next step in our execution process is to enter the loop and to proceed to symbolic execution. We execute $inst.3$ (line 3), here we have two cases : the loop is entered when the condition $i < T.length$ is true and the loop is not entered when the condition is not true. Thus, we build two additional formulas : $(max = 0) \wedge (i = 0) \wedge (i < T.length)$ and $(max = 0) \wedge (i = 0) \wedge (i \geq T.length)$. Next step is to execute the if statement $inst.4$ (line 4), here we have two additional branches and so two additional formulas where each formulas represent the abstract execution of each branch. The idea through this technique is to accumulate the conditions during symbolic execution and each time we enter the loop we add a new invariant. In the example above, we generate a new invariant candidate when we enter a second time the loop : the invariant φ_1. Naturally, we consider the disjunction of φ_0 and φ_1 as our new invariant candidate $(\varphi_0 \vee \varphi_1)$. We resume the symbolic execution of the program since φ_0 and $\varphi_0 \vee \varphi_1$ are not equivalent, we may generate a new invariant φ_2. This technique using only symbolic execution may not terminate. Thus, we resort to predicate abstraction.

- inst. $1 - 2$

$$\underbrace{(max = 0) \wedge (i = 0)}_{\varphi_0} \rightarrow$$

```
while(i < T.length){
if(T[i] > max) max= T[i];
i++;
}
```

- inst.3

$$(max = 0) \wedge (i = 0) \wedge (i < T.length) \rightarrow$$

```
if(T[i] > max) max= T[i];
i++;
while(i < T.length){
if(T[i] > max) max= T[i];
i++;
}
```

$$\wedge(max = 0) \wedge (i = 0) \wedge (i \geq T.length) \rightarrow$$

```
exit
```

- inst.4

$$(max' = 0) \wedge (i = 0) \wedge (i < T.length) \wedge (T[i] > max') \wedge (max = T[i]) \rightarrow$$

```
i++;
while(i < T.length){
if(T[i] > max) max= T[i];
i++;
}
```

$$\wedge(max = 0) \wedge (i = 0) \wedge (i < T.length) \wedge (T[i] \leq max) \rightarrow$$

```
i++;
while(i < T.length){
if(T[i] > max) max= T[i];
i++;
}
```

$$\wedge(max = 0) \wedge (i = 0) \wedge (i \geq T.length) \rightarrow$$

```
exit
```

$$\Longleftrightarrow \begin{cases} (max' = 0) \wedge (i = 0) \wedge (i < T.length) \wedge (T[i] > max') \wedge (max = T[i]) \\ \bigvee \\ (max = 0) \wedge (i = 0) \wedge (i < T.length) \wedge (T[i] \leq max) \end{cases} \rightarrow$$

```
i++;
while(i < T.length){
if(T[i] > max) max= T[i];
i++;
}
```

- inst.5

$$\underbrace{\begin{array}{l}(max'=0)\wedge(i'=0)\wedge(i'<T.length)\wedge(T[i']>max')\wedge(max=T[i']) \\ \bigvee(max=0)\wedge(i'=0)\wedge(i'<T.length)\wedge(T[i']\leq max)\wedge(i=i'+1)\end{array}}_{\varphi_1} \rightarrow$$

```
while(i < T.length){
if(T[i] > max) max= T[i];
i++;
}
```

- $$\varphi_0 \vee \varphi_1 \rightarrow$$

```
while(i < T.length){
if(T[i] > max) max= T[i];
i++;
}
```

- $$0 \leq i \wedge i \leq T.length \wedge \forall j.(0 \leq j < i \rightarrow T[j] \leq max) \rightarrow$$

```
while(i < T.length){
if(T[i] > max) max= T[i];
i++;
}
```

- $$0 \leq i \wedge \forall j.(0 \leq j < i \rightarrow T[j] \leq max) \wedge i \geq T.length \rightarrow$$

  `{}`

We can proceed again to symbolic execution to generate a new formula φ_2 for the loop, and then stop or go on accordingly. The problem with this plan is that it may not terminate this is why we resort to predicate abstraction to over-approximate the computing of the fixpoint for the loop and generate a set of candidate predicates that satisfies each formula generated during symbolic execution and using the previous steps. We need first to fix a set of predicates so we consider the following set of formulas $\{\varphi_0, \varphi_1\}$. Now, we generate a set of candidate predicates CP that satisfies both φ_0 and φ_1. Each predicate p in CP must satisfy $(\varphi_0 \vee \varphi_1) \rightarrow p$. We consider the following set of predicates for this example :

$$CP = \{\underbrace{i = 0}_{p_1}, \underbrace{0 \leq i}_{p_2}, \underbrace{i \leq T.length}_{p_3}, \underbrace{\forall j(0 \leq j < i \rightarrow T[j] \leq max)}_{p_4}\}$$

In general CP might be chosen by following heuristics, e.g., include all parts of the invariant candidate accumulated before the first unfolding of the loop, the loop guard, the weakest precondition computation and parts of the k^{th} iteration of the loop.
`For this example the set of all invariants must verify the formula below, this formula is the conjunction of predicates` p_2, p_3 `and` p_4 `in CP.`

$$\forall k, \forall j$$

$$\begin{cases} 0 \leq k < T.length, 0 \leq j < T.length \\ \forall k.(k < j \wedge T[k] \leq T[j]) \rightarrow max = T[j] \end{cases}$$

14.2.4 Conclusion

In this section, we present how to generate the set of predicates from the control flow graph of the program and how to infer the relations between them. First, we show how to handle execution traces that do not contain loops and we define the appropriate abstraction rule for them. Then, we take a trivial example that contain a loop and we enumerate steps how to generate the set of predicates by executing stepwise the loop. The abstract formula is

verified once a fixpoint is reached, we resort to predicate abstraction techniques to generate the formulas. In the next section, we take an example more intricate, the example of the simple bus where we have more properties to verify.

14.3 The Simple Bus Case Study

The Simple Bus case study is a well-known transactional level example, designed to perform also cycle-accurate simulation. It is made of about 1200 lines of code that implement a high performance, abstract bus model. The complete code is available at the SystemC web site [sys]. Figure 14.7 shows the bus structure. It uses a specific form of synchronization, where

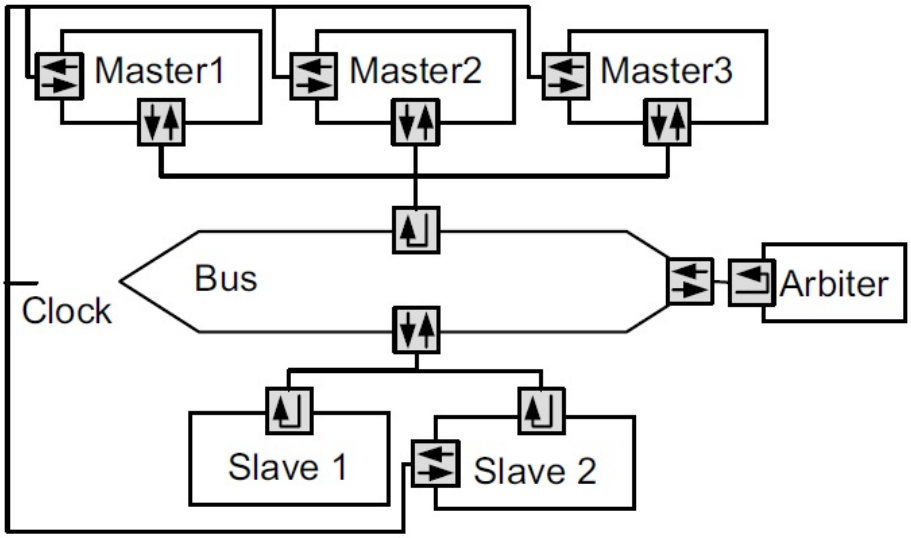


FIGURE 14.7 – *Simple Bus Structure*

modules connected to the *bus* execute on the rising *clock* edge, and the bus itself executes on a falling clock edge. Several *masters* can be attached to the bus. Each master is characterized by a unique priority, that is represented by an unsigned integer number. The lower this priority number is, the more important the master is. Each master communicates with the bus via an interface, which describes the communication between masters and the bus. Three modes of communication are possible : (1) *Blocking Mode* where data is transmitted through the bus in a burst mode without interruption even by a request with a higher priority. (2) *Non-Blocking* Mode where the master read or write a single data word. After the transaction is completed, the caller must take care of checking the status of the last request, which can

```
#include "simple_bus_arbiter.h"
template <class T>
simple_bus_request<T>
simple_bus_arbiter<T>::arbitrate(const sc_pvector simple_bus_request<T>*> &requests)
{
  int i;
  simple_bus_request<T> *best_request = requests[0];
  // highest priority: status==SIMPLE_BUS_WAIT and lock is set:
  for (i = 0; i < requests.size(); ++i)
    {  simple_bus_request<T> *request = requests[i];
      if ((request->status == SIMPLE_BUS_WAIT) &&
  (request->lock == SIMPLE_BUS_LOCK_SET))
{     return request;
} }
  // second priority: lock is set at previous call,
   for (i = 0; i < requests.size(); ++i){
    if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
        {   return requests[i];
        }
      }
  // third priority: priority
  for (i = 1; i < requests.size(); ++i)
    { sc_assert(requests[i]->priority != best_request->priority);
      if (requests[i]->priority < best_request->priority)
best_request = requests[i];
    }
  if (best_request->lock != SIMPLE_BUS_LOCK_NO)
    best_request->lock = SIMPLE_BUS_LOCK_GRANTED;
  return best_request;
}
```

FIGURE 14.8 – *Simple Bus Arbiter Code.*

be issued and placed on the queue (BUS, REQUEST), served but is not completed (BUS, WAIT), completed without errors (BUS, OK), or finally did not complete due to an error (BUS, ERROR). (3) *Direct Mode*, where the interface functions perform the data transfer through the bus without using the bus protocol. The *slave* interface describes the communication between the bus and the slaves. Multiple slaves can be connected to the bus. Each slave models some kind of memory that can be accessed through the slave interface. Two modes are possible : (*i*) Direct interface where it can perform immediate read or writing of data without using the bus protocol. (*ii*) Indirect interface where the slave can read or write a single data element. The functions return instantaneously and the caller must check the

status of the transfer. The arbiter is responsible for choosing the appropriate master when there is more than one connected to the bus. The arbiter performs the selection according to the following rules : (1) if the current request is a locked burst request, then it is always selected, (2) if the last request had its lock flag set and is again *requested*, then it is selected from the collection queue and returned, otherwise (3) the request with the highest priority is selected from the collection queue and returned.

This structure includes several SystemC components and nicely makes use of the principles of using SystemC at the transactional level. Besides some of the sample properties, e.g. liveness and safety, cannot be verified using simulation. They require the usage of formal techniques such as model checking.

To illustrate our method for predicate inference, we take as an example the code of the bus arbiter. The arbiter that manage priorities between the masters each time they want access to the bus. Figure 14.8 presents the arbiter code, the code includes three independent loops. To analyze the arbiter process, we need to analyze each loop independently and generate the set of abstract properties. We define abstract formulas to verify that each slave request is served and to verify which request to serve next.

The arbiter code is composed of three loops, our goal is to analyze each loop independently and generate the set of abstract formulas for each loop. To do so, we analyze for example the second loop and the method will be generalized for the other loops. We define the set of following tests generated from the loops conditions :

- $test_1(req) : (req \rightarrow status = SB\_WAIT) \wedge (req \rightarrow lock = SB\_LOCK\_SET)$
  $test_1$: `verifies if the parameter request is well defined and that it asks for a lock. It verifies as well, whether it was in the WAIT state.`

- $test_2(req) : (req \rightarrow lock = SB\_LOCK\_GRANTED$
  $test_2$: `verifies if the request for a lock is guaranteed.`

We denote the size of the requests table such that : $requests.size() = N$. The goal of the analysis is to generate an invariant for the loop, we start executing some iterations of the loop and then we generate an abstract formula that describes the computation. We resort to the technique of **loop unrolling** : a technique that consists in optimizing the program's execution speed by reducing or eliminating instructions that control the loop.

We resume the example of the arbiter, and more specifically the second loop. The second loop consists in browsing the table of requests and return the first request that satisfies $test_2$. We execute at most N iterations, we illustrate this on the following demonstration through different steps of execution.

- Step1 (i=0)

$$i = 0 \wedge 0 < N \rightarrow$$

```
if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
     {   return requests[i];
     }
for (i = 0; i < requests.size(); ++i){
   if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
     {   return requests[i];
     }}
```

- Step2 (i=0)

$$i = 0 \wedge 0 < N \wedge test_2(requests[0]) \rightarrow$$

```
return requests[i];
```

$$\wedge \underbrace{i = 0 \wedge 0 < N \wedge \neg test_2(requests[0])}_{\varphi_0} \rightarrow$$

```
for (i = 0; i < requests.size(); ++i){
   if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
     {   return requests[i];
     }}
```

- Step3 (i=0)

$$i = 0 \wedge 0 < N \wedge test_2(requests[0]) \wedge request = request[0] \rightarrow$$

```
exit;
```

- Step1 (i=1)

$$i = 0 + 1 = 1 \wedge 1 < N \rightarrow$$

```
if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
      {   return requests[i];
      }
 for (i = 0; i < requests.size(); ++i){
    if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
      {   return requests[i];
      }}
```

- Step2 (i=1)

$$i = 1 \wedge 1 < N \wedge test_2(requests[1]) \rightarrow$$

```
return requests[i];
```

$$\wedge \underbrace{i = 1 \wedge 1 < N \wedge \neg test_2(requests[1])}_{\varphi_1} \rightarrow$$

```
 for (i = 0; i < requests.size(); ++i){
    if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
      {   return requests[i];
      }}
```

- Step3 (i=1)

$$i = 1 \wedge 0 < N \wedge test_2(requests[1]) \wedge request = request[1] \rightarrow$$

```
exit;
```

- ..

- Step1 (i=N-1)

$$i = N - 2 + 1 \wedge N - 1 < N \rightarrow$$

```
 if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
      {   return requests[i];
      }}
 for (i = 0; i < requests.size(); ++i){
```

```
if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
  {   return requests[i];
  }}
```

- Step2 (i=N-1)

$$i = N-1 \wedge N-1 < N \wedge test_2(requests[N-1]) \rightarrow$$

```
return requests[i];
```

$$\wedge \underbrace{i = N-1 \wedge N-1 < N \wedge \neg test_2(requests[N-1])}_{\varphi_{N-1}} \rightarrow$$

```
for (i = 0; i < requests.size(); ++i){
   if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
     {   return requests[i];
     }}
```

- Step3 (i=N-1)

$$i = N-1 \wedge N-1 < N \wedge test_2(requests[N-1]) \wedge request = request[N-1] \rightarrow$$

```
exit;
```

- Step (i=N)

$$N \geq N \rightarrow$$

```
Exit;
```

We iterate the loop at most (N-1) times and at least once. Once we reach the N^{th} iteration, we exit the loop. The loop reaches a fixpoint when there exists just one request in the request table that satisfies $test_2$ and the set of requests before it don't, i.e, there exists an element i from the table *requests* such that $test_2(requests[i])$ is true and for each j such that $j < i$, $test_2(requests[j])$ is false. The goal behind the loop is to search for the first request that seeks access to the bus and serve it.

We resume the previous iterations, as we notice each formulas from $\{\varphi_0, \varphi_1, ..., \varphi_{N-1}\}$ presents a candidate invariant for the loop. Using predicate abstraction, we generate a set of predicates p such that $\varphi_0 \vee \varphi_1 \vee ... \vee \varphi_{N-1} \rightarrow p$. We proceed the same as in Section 14.2.3 : we first specify the set Ψ of formulas that satisfy the loop and constructed by iteration, then we generate a set of candidate predicates from the set Ψ. Let $k \in [0; N-1]$ be an integer, we suppose the following assumptions :

$$i = k \wedge \neg test_2(requests[0]) \wedge ... \wedge \neg test_2(requests[k-1]) \wedge \neg test_2(requests[k]) \rightarrow$$

```
for (i = 0; i < requests.size(); ++i){
   if (requests[i]->lock == SIMPLE_BUS_LOCK_GRANTED)
     {   return requests[i];
     }}
```

If we want to generalize the assumption above, we have just to quantify the logical expressions. We suppose that $\exists k \in [0; N-1]$ such that :

$$\underbrace{\neg test_2(requests[0]) \wedge ... \wedge \neg test_2(requests[k-1]) \wedge test_2(requests[k])}_{\Psi}$$

Formula Ψ is equivalent to : $\forall j \leq k, \neg test_2(requests[j]) \wedge test_2(requests[k])$. Otherwise, $\exists k \in [1; N], \forall j \in [1; N], j \leq k \wedge \neg test_2(requests[j]) \wedge test_2(requests[k])$
We verify by induction the previous formula :

❶ for $k = 0$ if $test_2(requests[0])$ then $j = k = 0$ and Ψ is true. If $\neg test_2(requests[0])$ then $k = j = 0$ and Ψ is true ;

❷ we suppose that Ψ is true for N ;

❸ we prove that it is true for N+1 ;

If it is true for N then :$\forall j, j \leq N \wedge \neg test_2(requests[j]) \wedge test_2(requests[N])$. Here we have two cases :

- case 1 : $test_2(requests[N+1]$, in this case,for each $k \leq N+1$, we have necessarily $\neg test_2(requests[j])$. Ψ is then true up to N+1.

- case 2 : $\neg test_2(requests[N+1]$ in this case Ψ is true until N which is previously verified. in this case we found an element from the table requests (N) that satisfies Ψ and all the forgoing elements don't verify $test_2$.

Interpretation of the analysis From the previous analysis, we generate the abstract formulas that describes the abstract behavior of the second loop in the arbiter example. As we previously explained the second loop extract from the table of requests the first request that verifies $test_2$. We also presented steps how to analyze the behavior of the second loop and how to use *the passage to limit* to generalize the final result since the size N of the table is unknown (how many times the loop is unrolled). As a result, we generate the formulas P_2 as follows : $P_2 : \exists\ i, \forall j, 0 \leq i < requests.size() \wedge test_2(requests[i]) \wedge 0 \leq j \leq i \wedge \neg test_2(requests[j])$ We use the same analysis to extract the abstract formulas for the first loop and the third one, we generate the formulas P_1 and P_3 defined as follows : $P_1 : \exists\ i, \forall j, 0 \leq i < requests.size() \wedge test_2(requests[i]) \wedge 0 \leq j \leq i \wedge \neg test_2(requests[j])$

$$
\begin{aligned}
P_3 :& \exists i, \forall j, 0 \leq i < requests.size() \wedge 0 \leq j < requests.size() \wedge \\
& i == j \vee requests[i] < requests[j] \rightarrow priority \wedge \\
& requests[i] \rightarrow lock \neq SB\_LOCK\_NO \wedge request[i] \rightarrow lock = SB\_LOCK\_GRANTED
\end{aligned}
$$

The analysis of the *SimpleBus* code shows then that we browse at most three times the list of queries in order to select what is the next request to be transmitted. The previous result is represented using the three logical formulas P_1, P_2 and P_3. Each formula represents a loop in the arbiter code.

CHAPTER 15

Conclusion

In this Part, we have presented an new automatic approach for verifying SystemC designs based on the SystemC waiting-state automata model (WSA). We show how to generate automatically the automaton for each component because the model used to be manually built in work of [YZM07]. We use a stepwise approach that starts from a well-defined formal semantics of a subset of SystemC language. Those semantics capture not only the structure of SystemC components but also the compositional behavior of the communicating components by including the semantics SystemC scheduler. In parallel, we proceed to symbolic execution of SystemC programs in order to generate the set of the execution traces of the program. During the symbolic execution, we generate the control flow graph (CFG). The nodes of this graph represent the basic commands and guard expressions of the process, and the edges stand for flow of control between the nodes. The control flow graph is annotated with exemplary of logical expressions called the path condition (PC). We combine the symbolic execution with the operational semantics and we call it the `extended` symbolic execution. The extended SE is used to generate a transition system that is syntactically conform to the semantics of the SystemC waiting state automata. Then, we explore predicate abstraction techniques to build automatically the SystemC WSA from the control flow graph genrated during the extended symbolic execution. Thus, we distinguish between two cases for program analysis :

first we consider programs without loops where we define our abstract formulas using the computation of the weakest preconditions to merge transitions and second we take a special case of programs with loops for which we define how to symbolically infer invariants using symbolic execution together with predicate abstraction. Finaly, we illustrate the approach on an example more intricate : The Simple Bus cas study.

In the next Part, we enumerate different applications of our framework based on the SystemC waiting-state model. First application is to propose a global framework based on the SystemC WSA to model and simulate embedded software/hardware systems. Second, we propose a conjoint work which brings together two research lines in our team unit : hardware/software co-verification of embedded systems and Computing worst case execution time (wcet). Hence, we propose a framework that symbolically execute the binary code on an abstract model of the processor using the Timed SystemC waiting-state automata[HM09] and applied an intelligent state fusion algorithm during symbolic execution as presented in [Ben11] to reduce the state space and give an exact estimation of the wcet. The third application propose to apply verification techniques notably model checking techniques to verify further properties on the SystemC waiting-state automata.

Part IV

Applications of the SystemC WSA Model

This Part introduces three applications of the SystemC waiting-state automata (*i*) modeling and simulation of programs, (*ii*) Hardware/software co-verification and (*iii*) Applying formal verification techniques to the SystemC waiting-state automata.

CHAPTER 16

Introduction

During *verification*, we assure that the software/hardware systems meet the requirements defined during specification. Verification includes also the functional requirements as well as architecture and design models, test cases, etc. In the previous section, we presented our approach for specifying and modeling an abstract representation of embedded systems described in SystemC. This representation is based on the SystemC waiting-state automaton. During the modeling step, most designers must ensure that their formal model satisfies the following specifications :

1. *A functional specification*, given as a set of explicit or implicit relations which involve inputs, outputs and possibly internal (state) information.

2. *A set of properties* that the design must satisfy, given as a set of relations over inputs, outputs, and states, that can be checked against the functional specification.

3. *A set of performance* indices that evaluate the quality of the design in terms of cost, reliability, speed, size, etc., given as a set of equations over inputs and outputs

4. *A set of constraints* on performance indices, specified as a set of inequalities.

The purpose of *validation* is to prove that the global system fulfills its required function when placed in its intended environment. There are two main techniques for system validation,

either static or dynamic techniques. The purpose with both techniques is to identify defects in the software. Static techniques are used to check and analyze representations of the system such as specifications, models and source code. Dynamic techniques (simulation) involve executing and analyzing an implementation of the software. But, *simulation* remains the main tool to validate a model, but the importance of formal verification is growing, especially for safety-critical embedded systems. During validation, designers must ensure that the abstract model can verify/detect some safety properties (including deadlock detection).

CHAPTER 17

Modeling and Simulation of SystemC Programs using the SystemC WSA

17.1 Introduction

The first step in static analysis of embedded systems is to give an abstract representation of the application in order to model the behavior of the system : this is the **modeling** process. The abstract model should be simple but also faithful to the initial system. Usually, the abstract model is a finite or infinite transition system where states represent different system locations and transitions represent the evolution from one state to another. The model should respect the tradeoff between an exact and proper approximation of the real system where most features of the system are preserved and having less complexity. Too many works for modeling embedded systems have emerged (examples of existing works are described in Part II), some of them are applied to purely software languages like C++ and Java and some others are used for software/hardware languages like SystemC and System verilog. Those models are used either to model the system at lower levels of abstraction where more details about system

behavior are defined or they are used to model the system at higher levels where details are hidden or are with less importance. The choice of the model must ensure the following criteria in addition to the previous ones : it should *i* be modular, *ii* be compositional, *iii* sustain refinement and *iv* support granularity.

Another important issue in system modeling is the model **validity**. Model validation techniques include specifically simulating the model using known inputs and comparing the model outputs with the system outputs. A model that is intended to be used for **simulation** is generally a mathematical model developed with the help of a simulation software. Simulation is used before the implementation of the final application in order to reduce the chances of failure to meet specifications, to eliminate system bottlenecks, to manage resource consumption, and to optimize system performances.

17.2 Modeling and Simulation with the SystemC WSA

In this chapter, we introduce an application of the the SystemC waiting-state automata to **model** and then **simulate** hardware/software systems. We propose a framework to design and symbolically execute parallel components of systems. The framework allows both to take into account the semantics of the SystemC WSA model and how to express the interactions between the automata.

Figure 17.1 is a schematic of the modeling/simulation approach using the SystemC WSA model. The approach starts from system specification where a detailed study of both the hardware and the software requirements is proposed. Then, regarding to the properties we want to verify, we precise at which level we want to model the system. It will be interesting if we propose a model that can represent the system at different levels of abstraction. We choose to model embedded systems using the SystemC waiting-state automata first because it is a compositional model and second because it models SystemC programs at both the TLM and the delta-cycle levels. The SystemC WSA is a transition system where states represent global states of the process and transitions are symbolic paths between states. Entry and exit conditions are defined over each transition. Once an abstract model is built for each component of the embedded system, we generate a global automaton for the whole system using a bottom-up approach. Next step is to validate the model which is already done during the process of building the automata as mentioned in Chapter 14. The validation is assured

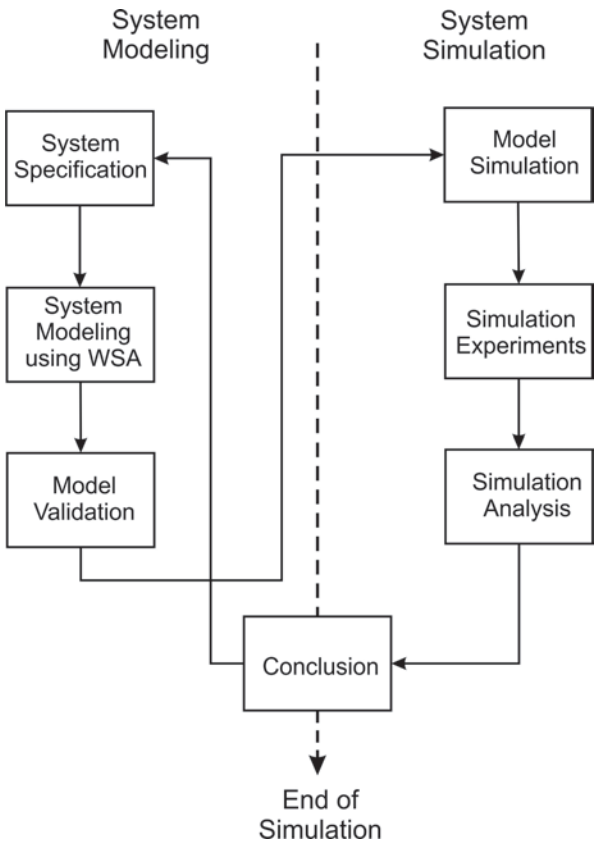


FIGURE 17.1 – *The Modeling and the Simulation process with the SystemC WSA*

during the step of applying symbolic execution and predicate abstraction to automatically build the abstract model.

Once the model is developed and then validated, we proceed to the symbolic simulation of the SystemC waiting-state automata (Figure 17.2). Hence, each SystemC automaton is considered as a **black box** where the internal behavior is already abstracted during the modeling process. We consider only the interactions between the automata and their environment. Thus, we can consider design executions as a set of abstract models which denote the observable states of the system and the communications between components. The latter representation provides the suitable environment to simulate symbolically systems modeled using the SystemC waiting-state automata. Hence, we affect symbolic values to global variables and events to allow the communication between the parallel processes. During symbolic simulation, we generate a graph that we call the **execution graph**. Before simulation, we choose our si-

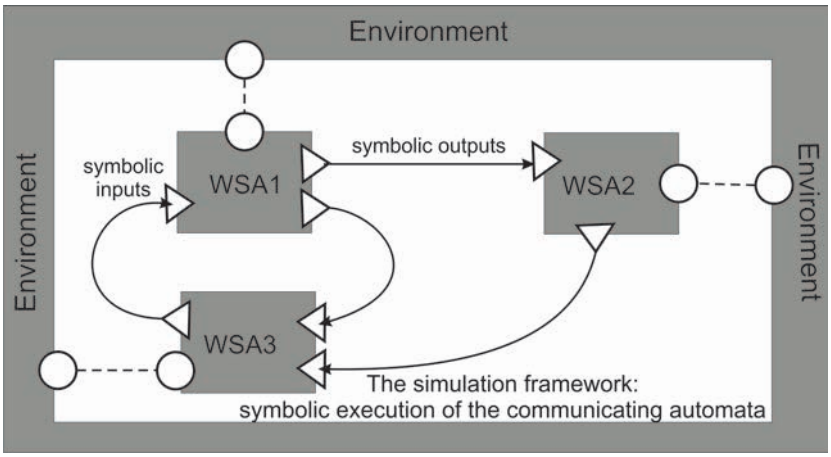


FIGURE 17.2 – *The Simulation Framework of the SystemC WSA*

mulation strategy, ie. we decide about different combinations of inputs and outputs, specify the properties to verify and select the appropriate experimental design. Then, we proceed to simulation Finally, we interpret and present the simulation results.

17.3 Summary

The main goal during complex system modeling is to propose an abstract representation of the system under-study that is faithfully conform to the initial system. We should ensure that most details about system behavior are covered by the abstract model. Besides, the model

should abstract the system under study independently from the details of its internal implementation. The SystemC waiting-state automata, as we previously proved, provides a proper and reliable representation of complex systems at both the delta-cycle level and the system level. Moreover, it provides a simplified and less complex representation of SystemC models since it represents only specific states in system behavior. The previous reasons substantiate the SystemC WSA model to be an efficient representation to symbolically simulate complex systems like in Figure 17.2. Indeed, we can easily and rapidly simulate the parallel behavior of embedded systems since we garantee first that the model is remarkably reduced during the step of the generation of the model. Second, we can symbolically simulate the model since it represents only symbolic values of inputs, so we don't need to use real values of inputs to simulate the parallel behavior of the global framework. Accordingly, the SystemC waiting-state automata provides the opportunity to symbolically execute and simulate complex systems in a dynamic abstract framework which greatly helps designers to validate and implement their application.

CHAPTER 18

Hardware/Software Co-verification (Worst Case Execution Time Estimation Workflow Based on the Timed WSA Model of SystemC Designs)

Real time systems are omnipresent in embedded systems and can be used either to optimize the process performance as well as to perform humanly uncontrollable activities. When involved in critical tasks they become hard real time systems thus the need to verify them. Ongoing approaches and tools based on dynamic and static methods or a combination of them, are used to either validate *functional* or *non-functional* properties. They are very few tools that verify both. The present application is a conjoint work [VPM11, Ben11] which brings together two research lines in our team unit at ENSTA ParisTech : hardware/software

co-verification of embedded systems and Computing worst case execution time (wcet).

Embedded systems are growing in complexity as they integrate different types of components including microprocessors (where pipelines and cache memory are becoming standard), DSPs, memories, embedded software, etc. thus the need of a global approach for systems description. One of the purposes of this global approach is to fill the gap between hardware description languages (HDLs) and traditional software programming languages. SystemC offers the hardware/software co-design capabilities while being able to model at different abstraction levels. SystemC can be seen as C++ with an added HDL layer so it provides a common development environment for software and hardware engineers. Combining these two features gave birth to a versatile language that takes the advantages from the object oriented programing paradigm but also the drawback of increasing the complexity of the verification process. In our approach, we propose a conjoint methodology based on the symbolic execution (*SE*) of an abstract model : the timed SystemC waiting- state automaton (TWSA) extracted from the SystemC model of the processor. One main advantage of using Timed WSA is that a specified model can be analyzed. This enables the avoidance of system failure by ensuring that certain requirements : functional as well as non-functional are fulfilled. Besides functional errors, a common cause of failures is timing violations, e.g. an inacceptable high response time of a critical piece of code or a control loop whose sample rate cannot be kept. Avoiding timing violations is only possible with knowledge about the worst-case timing of a task. The purpose of the worst-case exection time (WCET) analysis is to provide a priori information about the worst possible execution time of a piece of code before using it in a system. To be valid, the WCET estimates must be *safe*, i.e. guaranteed not to **underestimate** the execution time. To be useful, they must be *tight*, i.e. provide low over estimations. The WCET of a program is usually calculated in a two-stage process comprising a source code level (analyzing the program flow) and an object code level (analyzing the object code with respect to architectural factors like pipelines and caches). To obtain this WCET, three steps are necessary : (*i*) the task control flow analysis, which determines the possible program paths, (*ii*) the architecture effects analysis, which takes into account the various hardware components (CPU pipeline, instruction cache, etc) to produce timings for program paths, and (*iii*) the final WCET computation. In this second application of the SystemC WSA model, we propose a conjoint methodology based on the symbolic execution of the the abstract

model of the processor described in Timed SystemC waiting-state automata and the symbolic execution of the binary code of the program resulting in a control flow graph. Actually, computing WCET with static analysis is one of the main topics of our laboratory at ENSTA ParisTech. Bilel in [Ben11] defines a global approach that models the hardware using abstract state machines and use an intelligent algorithm for predicate abstraction to reduce the state space generated during symbolic execution of the model and the binary code. This approach is very interesting especially in term of the abstract states fusion algorithm. Besides, it gives good results in WCET estimation compared to existing approaches. But, we propose to use the WSA to model the hardware instead of the ASM to obtain more interesting results. Indeed, the Timed SystemC WSA is defined with less states compared to the ASM. Besides, it is generated independently from the hardware architecture compared to the ASM. More particularly, the Timed SystemC WSA is already annotated with timing information which makes the estimation of time execution more easier.

18.1 Introduction

Estimating the worst-case execution time of a program is a very important task, especially when you are dealing with real-time operating systems and programs, which have deadlines that have to be kept. Missing a deadline can have catastrophically consequences, because real time operating systems and programs are used in all types of time sensitive embedded systems, e.g. in medical equipment, cars, mobile phones and airplanes.
The purpose of *Worst-Case Execution Time* (WCET) analysis is to provide a priori information about the worst possible execution time of a piece of code before using it in a system. To be valid, WCET estimates must be **safe**, i.e. guaranteed not to underestimate the execution time. To be useful, they must be `tight`, i.e. provide **low** overestimations (Figure 18.1).
The worst case execution time approximation is not an easy task. Several things have to be considered, such as how to model the caching behavior to include it in the analysis and how to find the longest execution path in all the execution paths of the program. For dynamically changing systems, analysis methods have to be extended to run-time. For potential modifications of the running system, an online component must analyze, whether a modification is feasible with respect to the specified timing constraints and to the available hardware.
The WCET of a program is usually calculated in a two-stage process comprising a source code

level (analyzing the program flow) and an object code level (analyzing the object code with respect to architectural factors like pipelines and caches). In this work, we propose to model the hardware behavior using the model of the SystemC waiting-state automaton extended with time and then we symbolically execute the program on this model (**conjoint symbolic execution**). The generated graph is annotated with information about the current statement (either addition, multiplication, etc.), the task achieved during the pipeline stages (Fetcher, Dipatcher, etc.) and different cache states (cache-miss, cache-hit). The graph is also annotated with information about the time (estimation about the duration that may take a task). Worst

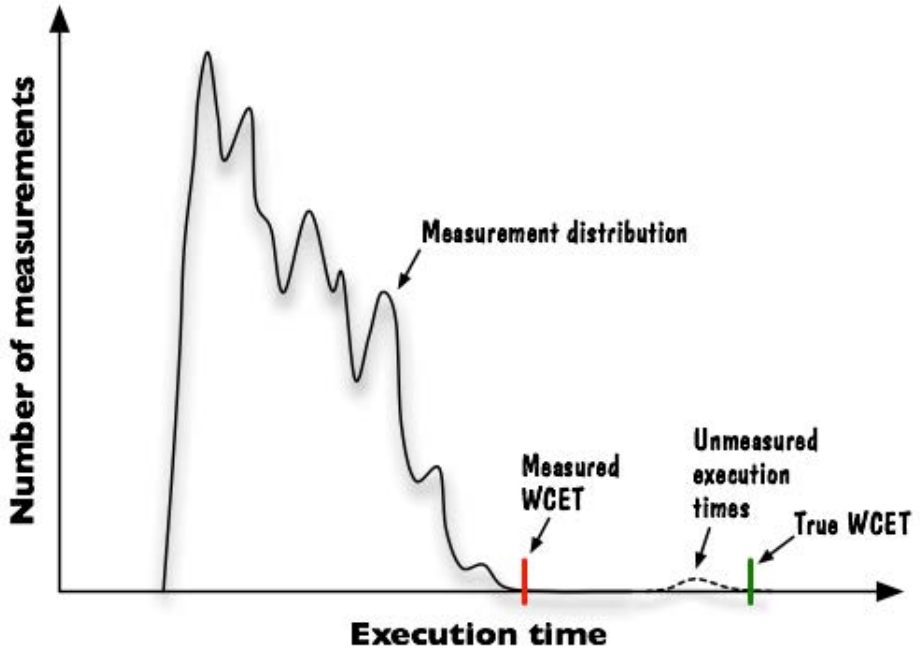


FIGURE 18.1 – *WCET Estimation*

case execution time is also a main topic in our lab as previously presented in the work of bilel [Ben11], our approach is based on the work in [Ben11] but we do more. In fact, the work of bilel [Ben11] uses the abstract state machine (ASM) to model the hardware (processor) and he has to define it manually and it differs from one processor to another, i.e, each time the hardware changes, the specification of the ASM may change too. Nevertheless, in the case of the SystemC waiting-state machine, the model is build independently from the architecture of the hardware, so, we don't need to change the specification we defined when the hardware changes. Besides, the SystemC waiting-state automaton is largly reduced in terms of state number compared to the abstract state machine since we consider only specific states of the

system description. This is why when we proceed to state fusion as in [Ben11], the algorithm we apply is much more easier to execute and then it takes less time to compute the WCET in a accurate manner.

18.2 Related Works

Static methods use formal verification technique in order to compute the worst-case execution time. The main advantage of static methods lies in their ability to provide complete coverage of execution traces.

Static methods use techniques for program analysis like symbolic execution (SE) to describe the behavior of the program. The approach of analyzing the intra-processor interactions or generating all the feasible paths by symbolic execution (SE) has been used with good results in [Lun02] but nevertheless the method suffers from the lack of a precise hardware model, using only a simulation of the latest with no correspondence between the real hardware and the timing model. This leads to a over-pessimistic time estimation. The lack of a good value domain and the indiscriminating state merging further contribute to the loss of precision.

The *OTAWA* method as introduced by Cassé and Sainrat [CS06], makes a first step towards adaptability as it uses a parametrized model of a generic platform that can address a variety of architectures. On the other hand, the process is fairly difficult and the model lacks precision while it fails to capture the precise behavior of the platform.

One of the leading WCET analyzer, aiT's AbsInt [CFW99, Wil04], is also evolving in this direction by looking to use a SystemC description in order to generate an abstract model. This technique was first developped by the AbsInt group in 1999, they use series of analysis to estimate the worst case execution time. First, they generate the control flow graph (CFG) from the binary code, then they proceed to value analysis in order to produce an over-approximation of the memory areas that will be accessed. The latter result is used to analyse the bahavior of the cache memory (cache-hit, cache-miss). Next step is to use the previous results to estimate the pipeline block at each execution point of the program. They use mainly abstract interpretation [CC77, CC92] to analyse their program, this technique is widely used in programs analysis. Finally, they use the previous analysis together with the source code analysis to generate and analyse the execution traces using a technique called *Integer Linear Programming (ILP)*. The use of ILP help to describe the program structure as well as the

set of the execution traces in a natural manner. Solving the set of constraints, that describes the program structure, using ILP help determine the worst case execution time. Although its effectiveness in WCET estimation, this technique has mainly the following drawbacks : (*i*) the overlap between different stages of program analysis may lead to many errors, (*ii*) the over-approximation of abstract values during abstract interpretation may affect the accuracy of the worst case execution time estimation and (*iii*) the necessity to properly model the processor (a component based representation of the Hardware, concrete and abstract semantics of the hardware, etc.).

Further objectives in the PREDATOR [prea] program are to guide the design of future architectures, making them more predictable in order to reduce the over approximation of the worst case behavior. The approach adopted in this work follows an autonomous paradigm, combining the exactitude and reliability of the hardware model being used (as we only translate the same code that was used to generate the system, we obtain a precise model of the architecture) with the ease of generating it. The advantages of our method are twofold, the ability to verify both functional and non-functional properties and the accuracy of the worst case behavior (in our case time related) estimation.

18.3 Modeling the Processor

Before we build the timed WSA, we need first to symbolically execute the code SystemC of the design. We obtain a detailed description of the design that contains intermediate states and elementary transitions : this is the control flow graph (CFG) where nodes present the statements of the process and edges are the transitions between these statements. In order to build the WSA, we consider only states that represent the wait statements of the process or thread. Those waiting states represent the synchronizing points between threads. Then, we resort to abstraction techniques to merge the transitions between each two waiting state.

Example

To illustrate our approach, we take the example of a block composed of the Icache, the Fetcher and the Decode (see Figure 18.2). We consider three modules representing the three stages. Each module has its own clock, its internal behavior and its local variables, but three

of them are communicating through events, signals, channels and shared variables. Further, modules are communicating through functions calls if we consider high level representations of SystemC. This is one of the main advantages of Timed WSA, since it provides an early system model plateform for software development. As it was previously explained, the Icache has no

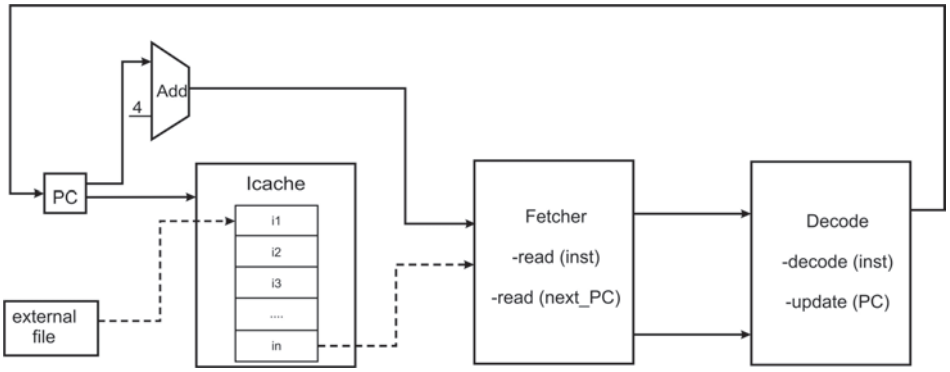


FIGURE 18.2 – *The Icache, the Fetcher and the Decode*

processes it contains only set of instructions. The Fetcher reads an instruction from the Icache and sends it with the next program counter $\mathcal{PC}$ to the Decode. The Decode determines the instruction type and updates the $\mathcal{PC}$. Figure 18.3 presents the CFG and the corresponding Timed WSA of the Fetcher. Starting from the CFG of the Fetcher, we consider only two special states where the Ftecher is waiting for its clock and waiting for the response from the Icache. Therefore, the Timed WSA of the Fetcher is composed of just two states.

18.4 WCET Estimation

Having the ability to adapt to any given architecture is becoming today a major concern, mostly because of the diversification and the growth in complexity of the platform used in the industry. Our approach is exploiting the popularity of SystemC designs in order to create a tool that is able to address any architecture described this way, by using a unified formal model, the timed SystemC waiting-state automata. In this section we will describe the way the WCET estimation technique uses the previously introduced model. In contrast with the approach presented in [BM09], our analysis starts directly from a SystemC design and is based on the Timed SystemC waiting-state automaton with the additional advantage that a first part of the abstraction is done at this level generating more compact states. Compared with the ASM based approach, the state explosion problem becomes less important. The main

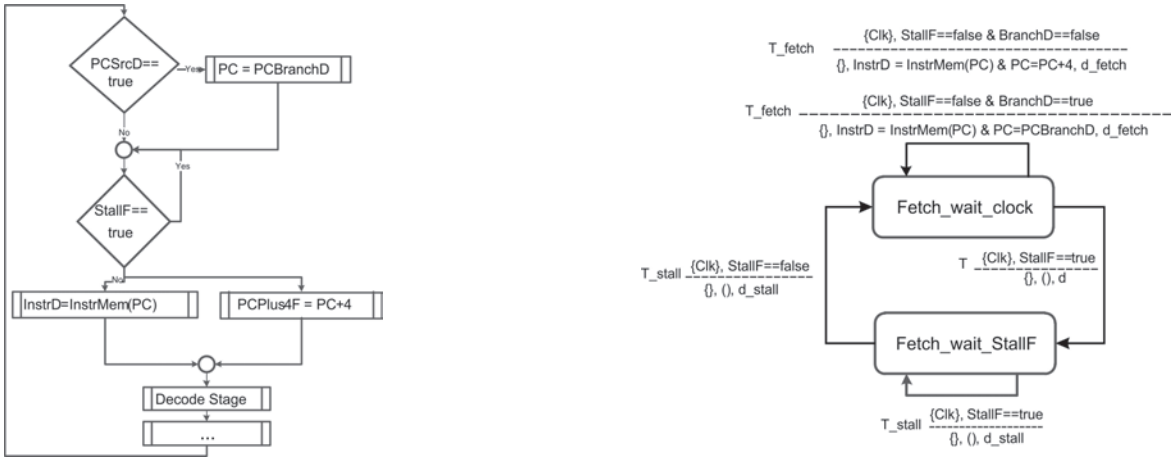


FIGURE 18.3 – *The CFG and the WSA of the Fetcher*

contribution yields from the new take on system modeling and the subsequent adaptation of the conjoint symbolic execution and state fusions criteria. Our analysis integrates a value analysis step that will exploit the binary of the program, followed by the conjoint symbolic execution (*SE*) of the processor model and the program information generated by the analyzer. The idea to symbolically execute the processor model has been successfully used in [Lun02] to determine all the feasible paths as well as to capture in detail complex processor behaviors like time anomalies or data hazards. Even if *SE* helps reducing the generated program states, by taking only feasible paths into account, it still generates a combinatorial explosion without providing a method of containment. We choose to handle it using our next analysis step - the *smart fusions*, described in [BBV08]. Other methods are under study, [THS09], and give the *SE* approach a new justification.

Figure 18.4 describes the global approach for WCET estimation : starting from a SystemC description of the Processor components, we extract the Timed SystemC waiting-state automaton for each unit of the processor and then we compose them using our symbolic composition/-reduction algorithms. Then, we proceed to the conjoint symbolic execution of the program together with our Timed WSA model as presented in [BM09]. We apply *value analysis* to the program that we explain more in detail in the next subsection to extract information about the instructions, the addresses, data values, etc. The final step is to execute *state fusion* on the control flow graph (CFG), this step is explained more in details in next subsection. This technique globally identify the *identical* states, i.e, they have either all the elements that are the same. In this case, we proceed to the fusion of strongly identical states. This will be done

by the *prediction module*.

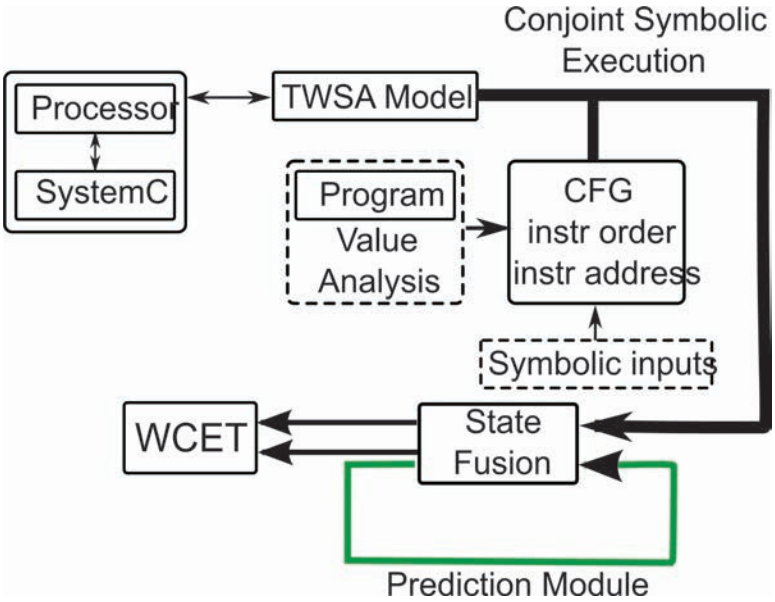


FIGURE 18.4 – *Global Architecture of the precise WCET estimation platform*

18.4.1 Value Analysis

The analysis of the program starts from the binary code giving information about the instructions order, their addresses, the loop counts and also serves to determine the memory areas that may be reached during the program execution. This result can be exploited in the cache analysis. This analysis is an integrated part of our tool and it was developed in collaboration with an industrial partner. The analysis is based on abstract interpretation and has a fast pass by default, but still giving the possibility to return at any moment at a certain program point and request for more precise approximations. This gives us a good enough result in a short amount of time while still keeping the possibility to be precise on demand.

18.4.2 Conjoint Symbolic Execution

In the following we will describe the main ideas behind conjoint *SE*. The TWSA gives us a precise (e.g. with regard to the time matter, even after the composition of the automata, the transition are labeled with the total duration of the respective actions) and yet compact (composition and then reduction of the automata) representation of the system. The model consists in a set of waiting states whose transitions represent the modification made when

certain conditions are met after receiving certain event notifications. Our technique consists in symbolically executing the TWSA processor model, as presented in Section 9.2 under a certain program run in order to generate all the reachable states of the processor under that specific context (the program whose WCET bound needs to be estimated). Symbolic execution consists in symbolically executing each instruction of the program meaning that every variable is replaced by a symbolic value. The succession of instruction generates a context and choices generate branches in the CFG that are accumulated into the path condition (pc). In the same way, the TWSA is symbolically executed, this time with the program instructions as inputs that will guide the evolution of the circuit and generate a new context and configuration that will also be accumulated in the pc in conjunction with the previous one.

Let Π_{SE} be a SE run defined as a succession of symbolic program points, $\Pi_{SE} = \{p_1, p_2, \dots p_n\}$ where

$$p = \left\langle (pc = Q), (x_i = R_i), (E_{out} = \dot{\bigcup} e^j_{out}), i = 1 \dots n \right\rangle$$

and

$$pc : \left\langle \dot{\bigcup}_{i=1}^{m} Q_i \wedge \dot{\bigcup}_{i=1}^{n} e_i^{in} \right\rangle .$$

The x_i, $i = (1 \dots n)$ are program variables whose values are expressed as formal expressions, R_i, over formal symbols and whose initial values are symbolic values α_i. Q represents an assertion specifying the conditions that must be verified in order to arrive in that specific program point and it's initialized to `true` in the initial state $p_0 = \langle (pc = true), (x_1 = \alpha_1), \dots, (x_n = \alpha_n) \rangle$ to which we add the *out* events, E_{out}, that were activated by the transition.

The pc works as a constraint accumulator. Each time a branching instruction is encountered in the code, the decision taken during the analysis is added to the pc. Similarly, when a branching occurs in the TWSA, the choice that lead to the next state (that translate into values taken by the different components of the automata), as well as the events that triggered it, are added in conjunction to the pc.The evaluation rule for assignment expression is obvious, the value of the symbolic variable at the left-side side of the assignment is replaced from now on with the value of the symbolic expression in the right-hand side of the assignment, after it's own evaluation - that might consist in replacing the variables used with their

corresponding symbolic expression. Let $p(pc)$ be Q, $p(x_i)$ be E_i and $p(\alpha \leftarrow \beta)$ be the old p where the value of α is changed to β. A special treatment is applied to conditional instructions that use the pc to explore all the possible scenarios. The expressions conjoined in the pc are of form $Q > 0$ where Q is a polynomial over symbolic values. Let R be this expression we thus have three possible cases : we can determine starting from the pc that the condition is always true, meaning $pc \supset R$ and $pc \not\supset \neg R$, therefore the execution will continue with the `then` branch analogue for the `else` branch or we can not determine if the condition is true or false, $pc \supset R$ and $pc \supset \neg R$, therefore the execution will continue along both branches, generating two new paths. In our case, the values of the variables represent values of the registers or their location. Operations on this values are isomorphisms. The classic *SE* needs only the current values of the program variables and a current instruction pointer, as well as the pc, in order to generate the symbolic tree. The significance of program variables in the case of the *SE* of the TWSA model would be the value of the registers, or their addresses, and we also need to store alongside the state of the processor (such as the current state of the pipeline). Therefore we need to keep track of the current waiting states that will give us the precise configuration of the processor

$$p = \langle \left(\dot{\bigcup}_{i=1}^{m} Q_i \wedge \dot{\bigcup}_{i=1}^{n} e_i^{in} \right),$$
$$\left(X = \bigcup_{i=1}^{p} R_i \right), \left(W = \bigcup_{i=1}^{q} S_j \right), \left(E_{out} = \dot{\bigcup}_{i=1}^{r} e_{out}^{i} \right) \rangle.$$

Execution of the Time Model of the Processor

Let $\{TWSA_{\mu P}, P, C(V)\}$ be the processor model, the program and the constraints on the program's variable respectively. The conjoint symbolic execution consists in symbolically explore all the feasible paths until we reach the ones corresponding to the worst case execution time. In order to do this we have a model that is able to take into account all the possible interactions at the interior of the processor, with regards to the execution time, guided by the instructions of the program which are revealed by the value analysis and applied on symbolic variables.

The temporal symbolic tree is defined as $Gs = \Pi_{SE}, TR, L$ where Π_{SE} are the symbolic

program points as previously defined, TR are the transitions between two program points, and L is the labeling function that will associate a time with each transition from a start program point ps until an end program point pe. Therefore we can introduce a time-accurate model as opposed to a cycle-accurate model [] that will enable us to reduce the combinatorial explosion that rises from the fact that we generate all the possible execution paths. For example if two consecutive states are identical we will not use all of them but we will rather generate only a relevant new state labeled with the time needed to obtain it.

The classic SE needs only the current values of the program variables and a current instruction pointer, as well as the *pc*, in order to generate the symbolic tree. The significance of program variables in the case of the SE of the Timed WSA model would be the value of the registers, or their addresses, and we also need to store alongside the state of the processor (such as the current state of the pipeline). Therefore we need to keep track of the current waiting states that will give us the precise configuration of the processor.

Symbolic State

We can further refine the notion of state that we use to capture all the information needed to the evolution of our system that is compact enough but can capture all the needed informations. The classic *SE* needs only the current values of the program variables and a current instruction pointer, as well as the *pc*, in order to generate the symbolic tree. The significance of program variables in the case of the *SE* of the Timed WSA model would be the value of the registers, or their addresses, and we also need to store alongside the state of the processor (such as the current state of the pipeline). Therefore we need to keep track of the current waiting states that will give us the precise configuration of the processor

$$p = \left\langle \left(\dot{\bigcup}_{i=1}^{m} Q_i \wedge \dot{\bigcup}_{i=1}^{n} e_i^{in} \right), \left(X = \bigcup_{i=1}^{p} R_i \right), \left(W = \bigcup_{i=1}^{q} S_j \right), \left(E_{out} = \dot{\bigcup}_{i=1}^{r} e_{out}^{i} \right) \right\rangle .$$

The execution algorithm consists in the following steps that are executed while we have not reached a final state.

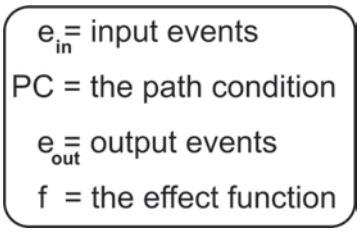


FIGURE 18.5 – *The symbolic state*

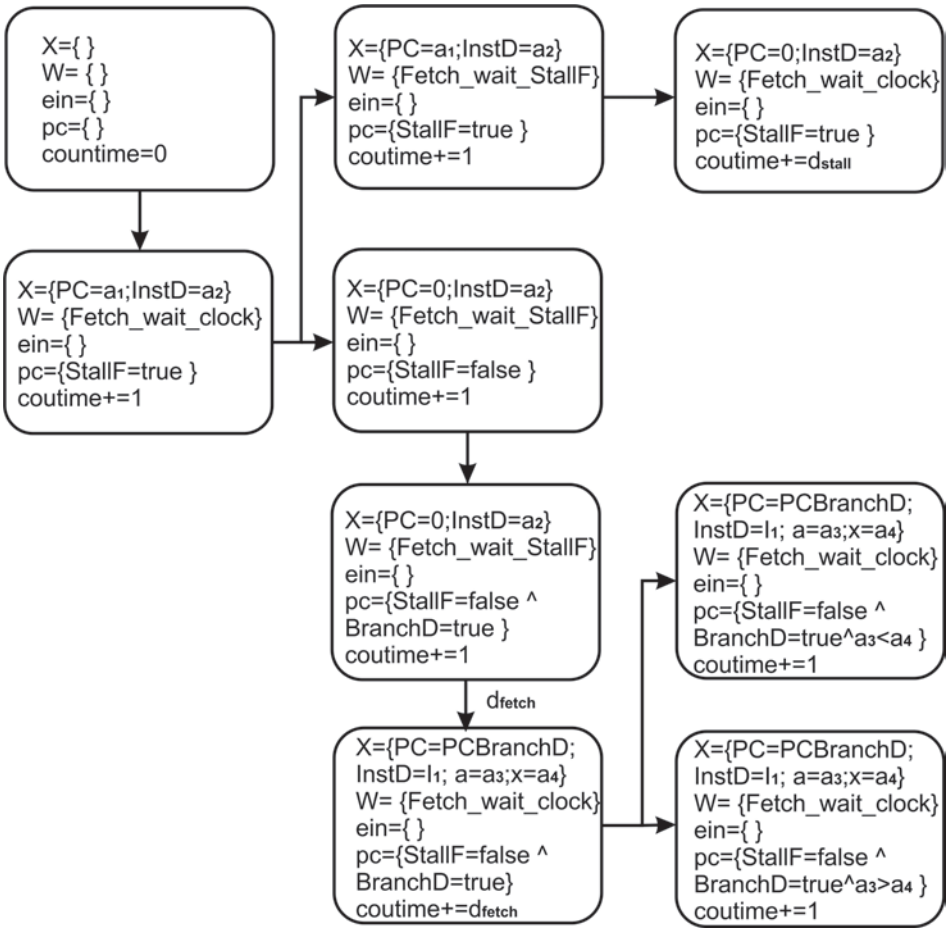


FIGURE 18.6 – *An example of symbolic tree*

Algorithm [BM09]

1. Start from the initial state : where all the components have the unknown value and pc is set to $true$
   $Init\_state = C_0 \leftarrow \{< (pc = true), (X = \perp), (W = \perp), (E_{out} = \perp) >\}$
2. For every variable that we encounter and that we do not have the exact value, assign a symbolic value
3. Activate the first waiting state of the Timed WSA model and then add the predicate P and the e_{in} to the pc
4. Add e_{out} to the system state
5. Apply $F(\overline{\alpha})$, a symbolic expression representing all the modification to be made, to the previous state of the Timed WSA model of the processor
   $C_i = \{p(X(\overline{\alpha})) \leftarrow F(\overline{\alpha}), \forall p \in C_{i-1}\}$
6. Add the generated states to the collection of next states to be executed
7. Add the duration of the transition to the global time
8. Repeat from point 2. until the collection of next states is empty

An implementation of a similar WCET estimation method based on abstract state machines gave promising results [BM09].

18.4.3 Intelligent States Fusion

One of the major drawbacks of the SE comes from it's quality of generating every feasible path, that for a real-life industrial program generates a combinatorial explosion that is not obviously containable. What still remains challenging today is to handle this explosion while still remaining precise enough. This translates to finding a way of eliminating some of the states, and we choose the technique of states fusion that will try to generate an abstract state capable of capturing the respective states features, with regards to the goal, but remain as compact as possible.

It has been proven in [BM09] that because of the finite number of states that a processor can

have and because of the constrains generated by the execution contexts at a certain point we will have states that regardless of the different history, will generate identical or very similar new states. One major step in having precise fusions is to determine when to make them and what changes to apply. States can be of two types as mentioned in Figure 18.7 : identical, meaning that they have either all the elements that are the same, in this case we can suppose that an eventual fusion will not impact the precision of the analysis, or similar, some of the components are not the same so we proceed to another analysis to determine to which extent they are different. Therefore similar states can be strongly or weakly similar, meaning that the impact of the fusion will be acceptable or not. For the instant this estimation is done dynamically by our prediction module. Its goal is to evaluate the impact in the future of a fusion by unrolling the tree for several steps (generally equal to the pipeline depth), continuing the execution along the paths before and after fusion and comparing the result. Further details about this technique can be found in [BM09].

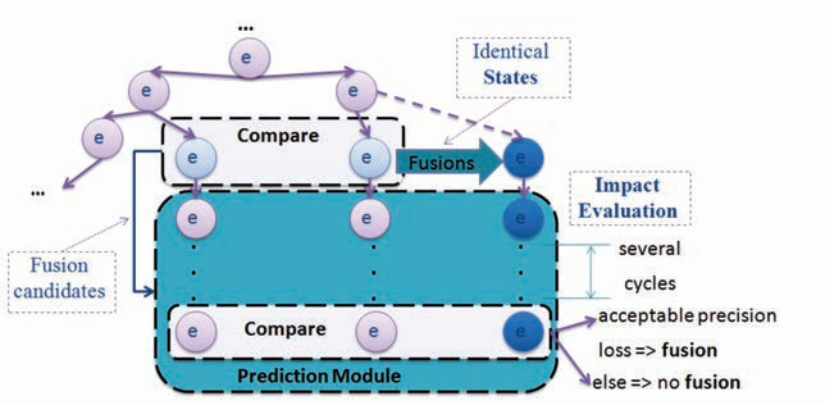


FIGURE 18.7 – *The Dynamic Fusion-snapshot of the Prediction Module*

18.5 Conclusion

The world of embedded software is no longer integrating simple hardware/software, therefore critical systems are becoming more and more difficult to prove and certify. The growth in complexity and variety increases the need of versatile analyze methods and adapted tools, that can easily and as costless as possible deal with a large panel of architectures. To this end we present a novel approach that is able to respond to the evergrowing demands and to place itself into a real industrial context. Our platform ultimately addresses both functional

and non-functional properties verification of systems and it could be used to compute several worst case behaviors,WCET being just one of them.

CHAPTER 19

Applying Verification Techniques to SystemC WSA

19.1 Anomalous Behaviors

The specific constraints that must be satisfied by embedded systems, such as timeliness, energy efficiency of battery-operated devices, dependable operation in safety-relevant scenarios, short time-to-market and low cost, particularly in consumer products, coupled with the never-ending pressure to increase the functionality, lead to an enormous growth in the complexity of the design at the system level. In this chapter we investigate the notion of design *complexity*. We argue that it is not the embedded system, but the models of the embedded system that must be simple and understandable.

The introduction of appropriate levels of abstraction in modeling and the transition between them help to reduce the emerging complexity of today embedded systems. In fact, those models focus specifically on the relevant properties and omit the irrelevant details, which leads to a simpler representation of the evolving embedded system.

19.1.1 Introduction

Abstraction from low levels to high levels and vice versa, should garantee some fundamental properties especially those related to interactions between concurrent processes. For example, in the case of the SystemC waiting-state automata where the degree of granularity achieved the delta-cycle level, where interactions are more and more intricate. The proposed model is supposed to provide the guarantee of some critical properties, among others, we cite two properties here : *liveness* and *determinism*.

19.1.2 Liveness and Determinism

The liveness property is probably the most common one in the field of formal verification. It simply states that the implementation must be deadlock-free, and in SystemC modeling, it means that there is no causality waiting cycles between processes, i.e two threads should not wait for each other at the same time. For instance, in the FIFO module, the producer and the consumer musn't wait for each other simultaneously. Indeed, a causality cycle can be triggered by a corner case condition in the behavior of the composition of a system of *asynchronous* components. In a simulation, one can observe a causality cycle when a computation does not stabilize to specific output values in an instant and keeps re-triggering itself. In our semantics rules, a causality cycle occurs when it never gets to the next delta-cycle.

Determinism is a very critical property for hardware design. In SystemC modeling, a deterministic SystemC model should ensure that the behavior is independent of the order of internal process executions. It stresses in particular on the determinism at the level of delta-cycles, because nondeterministic behaviors are caused by competitions when multiple processes are accessing shared resources at the same time, and such competitions usually occur within a single delta cycle.

As an example, consider another implementation of FIFO (Figure 7.2) : the producer/consumer does not wait for the other to release/fill the buffer when it is fullempty, and instead,

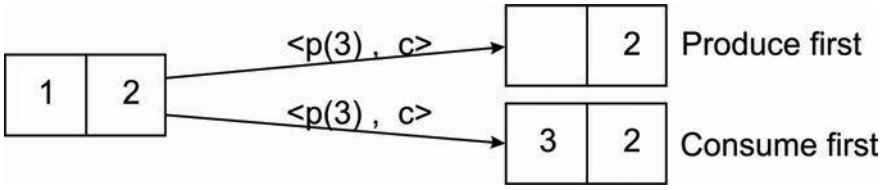


FIGURE 19.1 – *Non-deterministic behavior*

they just return a write/read failure. It is clear that there might be a competition between the producer and the consumer : if at the beginning of a delta-cycle, both producer and consumer are allowed to operate on the buffer (both *p_clock* and *c_clock* are present), we might have different results, as shown in Figure 19.1 ($< p(3), c >$ means that both *p_clock* and *c_clock* signals occur at the same instant, i.e. the producer and the consumer are boty ready to be executed. The producer is ready to put the data 3 into the buffer). In this case, we have too different scenarios :

- If the producer will execute first, then the data 3 will be lost since the buffer is already full and consumer will read data 1.
- If the consumer will execute first, data 1 is read by the consumer and then the producer will add data 3 to the buffer.

If the buffer is full, producing first will cause the new product to be discarded, while consuming first will cause a successful writing to buffer.

Non-determinism is hard to detect through simulation, because the SystemC scheduler will fix an execution order for process, though semantically, it should not be fixed. For instance, it may always execute consumer first, then the simulation will return the same result with the same input and we cannot see the non-deterministic behavior from the simulation.

19.2 Applying Model Checking Techniques to SystemC WSA

19.2.1 Introduction

Model checking [ECP99a] is a technique to automatically verify finite state concurrent systems. It consists in proving if an abstract finite model M defined in a certain logic verifies a property p expressed in the same logic. Model checking has been successfully adopted for

both hardware and software verification. Without loss of generality, the core techniques of model checking rely on the analysis of reachability property of the set of states. Therefore, it is required that the states and the corresponding transitions of the design under verification should be clearly defined. For hardware, the states are the valuation of the flip-flops and the transitions are the combination logic in the circuit ; for software, they are the valuations of variables and the statements in the program, respectively.

As shown in Figure 19.2, we apply model checking on SystemC using the SystemC waiting-state automata as follows : First, we need to translate the SystemC WSA with timed language constructs into an intermediate model so that we can easily apply model checking techniques. We can use either **timed automata** [TA90, AD94], a transition system annotated with a set of real-valued variables called clocks that increase synchronously with time and associates guards and update operations with every transition, or existing abstract models like **Kripke structures**. Then, we use temporal logics to express the property we want to verify on the abstract model.

Many approaches apply model checking techniques to verify SystemC. These approaches differ on the models they use to interpret the SystemC semantics. Nevertheless, they either fail to not handle all SystemC constructs like [eD05a], or are bound not to scale up specially when the system require non-deterministic behavior [MFM06]. To deal with all these limitations, we propose an efficient model checking approach based on the SystemC waiting-state automta because :

1. the state explosion problem is already reduced in the waiting-state automaton model since we consider only specific states to extract the automata,

2. we don't need to model separately the SystemC scheduler since it is already included in our formal semantics for SystemC. Thus, the scheduling of the concurrent behavior of the system can not influence the execution paths of the design and so the waiting-state automata,

3. the number of states in the waiting-state automata to explore is enormously reduced,

4. our predefined semantics supports all SystemC constructs and communication mechanisms (channels, signals, etc.),

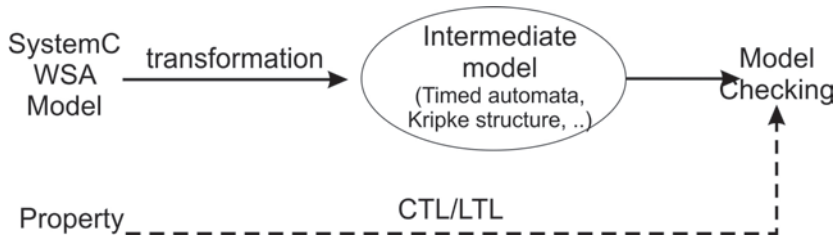


FIGURE 19.2 – *Applying MC to SystemC WSA*

5. signals and variables with large domain, e.g. integers, are already taken into account and present no problem in our modeling approach since they are **symbolically** modeled,

6. to deal with unbounded loops that are not supported in some model checking techniques like the approach of [eD05a], we used predicate abstraction as shown in Section 14.2.

19.2.2 Checked properties

In the following, we enumerate the main properties to verify on the SystemC waiting-state automata.

- Safety property : it concerns variables values which have to satisfy certain constraints. This is already reflected during the symbolic execution of SystemC designs, since we use symbolic values of variables instead of real ones. But, we need to prove the previous assumption using model checking. We express this property as a set of assertions defined over the set of predicates used in transitions. the failure of those assertions involves refinement of the initial model. If we take the example in Section **??**, we need to verify after symbolically executing the code that all the element of the table are sorted.

- Transaction properties (TLM) : check whether a request or a response is (in)valid or whether a transaction is successful. If we take the case of the simple bus (Section 14.3), we prove that each data written into the bus arrives to its destination without loss of information.

- System level properties : check on the order of occurrence of event notifications and the order of transactions. This property concerns the order of notification of input events in the abstraction rule (Section14.2.2) and how to manage the set of requests in the simple bus case study (Section 14.3).

We express the previous properties as follows :

`Safety property` :

A transition from a state σ_i to σ_{i+1} is called **safe** when it has no assertion failure. It is written $safe(s_i, s_{i+1})$. Thus, we need to verify that each execution trace defined in Section **??** satisfy the property defined as follows :

$$allSafe(\sigma_0, \sigma_n) = \bigwedge_{0 \leq i \leq n} safe(\sigma_i, \sigma_{i+1})$$

The relation **allSafe** is used to express that all the consecutive states from σ_0 to σ_n are safe. Thus, we say that a state in the SystemC WSA is `reachable` iff all the execution traces that lead to that state are safe. We prove this by induction over each execution trace.

`Transaction property` :

We label each write transaction into the bus as M_WRITE_DATA and each read transaction from the bus as S_READ_DATA. The checked property consists in verifying whether the number of data written into the bus is equal to the number of requets read from the bus. We express it as follows :

$$assume\ number\_of(M\_WRITE\_DATA) \leq number\_of(S\_READ\_DATA)$$

`System-level property` :

The first property verifies whether the abstraction rule respects the order of notification of the input events. The second property verifies that each request in the table of requests verifies at least one property in $\{P_1, P_2, P_3\}$.

for each input event (Section 14.2.2) $e_{in} \in E \Rightarrow F_E = true$

for each request (Section 14.3) $Rq \in requests[n] \Rightarrow P_1 \vee P_2 \vee P_3 = true$

Part V

Conclusion and Prospects

This part enumerates the main contributions of this thesis and the possible prospects for the approach.

CHAPTER 20

Conclusion and Prospects

20.1 Results and Discussion

Modeling reactive systems, critical systems or embedded systems is a very important issue today, since it facilitates the verification/validation step that comes after. The increasing complexity requires more design efforts and choosing the right architecture to guarantee system performance and reliability requires large design space exploration.

In this thesis, we have presented an automatic compositional approach for verifying SystemC designs based on the SystemC waiting-state automata (WSA). We prove that the abstract model is faithful to the simulation semantics of SystemC at both the transaction level, where details about the system implementation are hidden from the system description, and at the delta-cycle level, where verification of temporal properties and the interactive behavior of the system components are crucial (an overview about the global approach as well as a comparison with existing approaches for SystemC modeling are presented in Part II).

The process of the modeling of SystemC designs is clear : First, we automatically build an

abstract representation of SystemC components using the SystemC WSA model. The model was firstly proposed in [YZM07], where authors need to manually build the automata for each component. In this thesis we propose an automatic stepwise framework to generate and extract an automaton for each component. Next, we combine the automata generated for each component in a bottom-up approach in order to build the automaton for the global system. In Chapter 8, we use algorithms for symbolic composition and symbolic reduction as defined in [YZM07], where different concurrent communications between processes are taken into account. During symbolic composition, a set of concurrent states is generated, where possible generation of **unsafe** states is avoided because the presence of unsafe states creates a deadlock situation in the whole system. During symbolic reduction, we distinguish between different transitions generated during symbolic composition. We propose to remove impossible transitions, to keep safe transitions and reduce redundant and reducible transitions. The goal behind the symbolic composition and reduction is to take into account possible and reliable interactions between system components.

In Chapter 9, we enumerate different possible extensions of the SystemC waiting-state automata with parameters. Those parameters are used either to detect anomalies due to concurrent access to shared resources or to verify temporal properties about the execution time. First, we resume the extension proposed in [YZM07], where authors propose to extend the abstract automata with counters. Counters are used to impose more constraints about the system behavior, i.e, they are used to detect infinite behavior of the system during parallel composition. Thus, this parameter is used more to verify functional properties about system behavior. Regarding to the verification of the non-functional properties of the system, we propose in latter work to extend the automata with time properties which was missing in [YZM07]. Verifying timing properties is essential in order to study the dynamic behavior of real-time embedded systems. Timing properties include strict deadlines, periodic execution of processes and external event recognition based on time of occurrence. We define two types of time parameters : the starting time and the duration for each transition. We also propose how to infer relations between different parameters with respect to the occurrence of different events defined on transitions and with respect to the execution time.

In Chapter 10 and later in Part III, we propose an automatic framework to generate the SystemC waiting state automata from SystemC designs which was missing in [YZM07],

where authors need to manually generate the automata from SystemC designs. In Chapter 10, we propose how to generate the automata from different processes where we distinguish between threads and methods in SystemC. Methods are uninterruptible process and have no wait statements, so their corresponding automata have only one waiting state. While, threads have one or more wait statements. Each wait statement represents a state in the abstract model of the process. In Part II, we propose a stepwise framework to automatically generate the SystemC waiting state automata from SystemC designs. To build the automata, we need first to define clear and efficient semantics of SystemC to capture the reactive behavior of SystemC components. We use the structural operational semantics to present semantics of a subset of SystemC. The operational semantics can increase the correct understanding of a language and gives the possibility of formal reasoning. The formal semantics capture the (i) synchronous and asynchronous process composition of SystemC components, (ii) all levels of abstractions for communications, and (iii) relation between simulation correctness and logical correctness.

In Chapter 13, we propose to use symbolic execution [Kin76, Dar88] to present the effect of executing statements on system variables and events. The symbolic execution generates the control flow graph (CFG) of the program where the nodes of this graph represent the basic commands and guard expressions of the thread, and the edges stand for flow of control between the nodes. The CFG is annotated with exemplary of logical expressions defined over variables and the path condition (PC) defined over conditional statement which represents an accumulation over a corresponding path of execution. We call it an *extended symbolic execution* because the symbolic state is not only defined over statements and the path condition but also defined over the input and the output events of the environment. During symbolic execution, we generate the set of all the execution traces. Besides, we combine the operational semantics, previously defined, with the symbolic execution.

Since the symbolic execution is itself not approximative, but as precise as possible. Instead, the necessary approximation is performed by explicit *abstraction* operations, which make use of an arbitrary, finite set of predicates over the variables of the program. The WSA model is then build using mainly abstraction techniques. Thus, the problem of building the WSA is then reduced to the problem of guessing potentially useful predicates. This technique is described in details over several examples in Section 13.2, we analyze the case of programs

without loops and then we take the example of programs with loops. In the case of programs without loops, we define an abstraction rule for events abstraction and predicate abstraction. For predicate abstraction, we define how to generate the weakest-preconditions from each subset of predicates and functions. In the case of programs with loops, we propose to execute stepwise the loop until we reach a fixpoint. We then propose to use heuristics in order to fix a set of candidate predicates from the abstract formula previously generated. Next, we study the simple bus program and we illustrate different steps for predicate abstractions throughout the arbiter code.

Finally, we present three applications of the SystemC waiting-state automata to verify both functional and non-functional properties of hard real time systems that have strict time constraints. First in Chapter 15, we propose a global framework to model and then simulate SystemC programs using the SystemC WSA. We propose to symbolically execute the SystemC waiting-state automata in order to symbolically simulate the application under study without really executing it on real inputs and to avoid exhaustive simulation due to unbounded test cases. Second in Chapter 16, We propose a global framework based on the SystemC waiting-state automata in order to give a precise worst-case execution time estimation. Thus, in this work, we are focusing on the Timed SystemC WSA processor's construction followed by the conjoint symbolic execution of the architectural model and the running program. In this sense we believe that the adaptability of a tool to ever changing architectural models is just as important as a tight estimation of the worst case behavior. Moreover, given the upcoming certification standards, being able to verify the correctness of the model that the analysis is based on is a further reason to generate it directly from the HDL code that served to create the system. Later in Chapter 17, we propose to use an approach based on the SystemC waiting-state automata to detect anomalies due to concurrent behavior of parallel processes and abstraction at different levels. We study two main properties : liveness and determinism. In Section 17.2, we propose to use model checking techniques on the SystemC WSA model to study for example the reachability property of the set of states. We propose to use existing abstract models that we generate from the SystemC waiting state automata. We then propose how to express different properties in a formal way, we stress on three main properties : the safety property, the transaction property and system-level property.

20.2 Prospects

The approach we propose in this thesis is basically used to model hardware/software embedded systems in order to verify targeted functional and non-functional properties about these embedded systems (as mentioned in Part IV). But, it can be extended to handle larger systems and to verify further critical properties in embedded systems : this is the main purpose of my future work for this thesis.

Future works include also case studies of large examples besides the long-term compilation work. Also, the model itself demands probably further refinement so as to fit well in real cases. Furthermore, we intend to make the process of building waiting-state automata fully automatic using further abstract techniques for programs analysis. We also plan to perform our formal semantics to handle all SystemC constructs.
Another line of future work, which is more speculative, concerns different techniques for validation that we used during this thesis like predicate abstraction and techniques that we intend to use in future works. One could for example investigate the use of a real application of model checking techniques to arrive at useful relations inference between predicates, to simulation in order to test a real application of the approach, etc. Below we describe the techniques of systems validation and we compare possible extensions of them in our work : what we achieved till now and what we want to do in future works.

Test The dynamic test consists in submitting the program with a set of inputs and run it to verify if it respects the specification. The test is therefore applied at the end of the development of the global system, to ensure that this part is correct. We can use it to validate each component of the system separately, and can also be used to validate the whole system [MS04]. It is the most common technique used in industry to validate programs, because of its modular feature for validation. We distinguish between two testing cases : the functional and the structural testing. The functional testing consists in submitting inputs to the program and then verify if the program meets the initial specifications. The structural testing consists in analyzing the program code in order to determine the minimal and the adequate set of tests that cover a maximum of possible behaviors of the program under study. So, *testing program can be used to prove the presence of bugs, but never their absence* (Dijkstra, 1974).

Applying simulation for embedded systems is a challenging task because they are **hete-**

rogeneous. In particular, most contain both software and hardware components that must be simulated at the same time. This is the **co-simulation** problem : the basic co-simulation problem is to satisfy and verify two conflicting requirements :

- to execute the software as fast as possible, often on a host machine that may be faster than the final embedded CPU, and certainly is very different from it ;
- to keep the hardware and software simulations synchronized, so that they interact just as they will in the target system.

One approach, often taken in practice, is to use a general purpose software *simulator* (based, e.g., on VHDL or Verilog) to simulate a model of the target CPU, executing the software program on this simulation model. Thus, different models can be employed depending on the abstract level, with a trade off between accuracy and performance :

- *Gate-level models*
  These are reliable only for small validation problems, where either the processor is a simple one, or very little code needs to be run on it, or both.
- *Instruction-set architecture (ISA) models augmented with hardware interfaces*
  An ISA model is a standard processor simulator (often written in C) augmented with hardware interface information for coupling to a standard logic simulator.
- *Bus-functional models*
  These are hardware models only of the processor interface ; they cannot run any software. Instead, they are configured to make the interface appear as if software were running on the processor.
- *Transaction-based models*
  These convert the code to be executed on a processor into code that can be executed natively on the computer doing the simulation. Preserving timing information and coupling the translated code to a hardware simulator are the major challenges.

When more accuracy is required, and acceptable simulation performance is not achievable on standard computers, designers sometimes resort to *emulation*. In this case, configurable hardware emulates the behavior of the system. Another problem is the accurate modeling

of a controlled electromechanical system, which is generally governed by a set of differential equations. This often requires interfacing to an entirely different kind of simulator. Today, safety in component based systems require a higher confidence criteria than tests.

Unlike the previous models for systems simulation, our approach is a suitable framework for systems validation using tests for many reasons : (*i*) because the SystemC waiting- state automata represent both the hardware and the software parts of the system so we do not need to validate each part independently, (*ii*) the SystemC waiting-state automata descibes the system at lower levels like the delta-cycle level as well as the higher levels like the transaction level, in this case the simulation is amenable to verify further properties and to give more precise results, and finally, (*iii*) the SystemC waiting-state automata contain information about both the non-functional and the functional behavior of the system components, this may help avoid anomalies generated during simulation such as deadlocks and gives an accurate information about timing constraints such as the worst-case execution time (WCET).

Model Checking Model checking [EC81, QS82] is a technique based on three concepts : a model of the system we study, a specification expressed as a property (a temporal property) and the algorithm used to verify if the model respects the specification. The model is defined using a kripke structure [MCBG88] that described the whole possible states of the system, the transitions between different states and the the atomic properties that each state must satisfy. The complex properties are given in temporal logic which can express causal relations between states. Model checking covers the set of all states of the system, i.e, all execution cases are considered. Moreover, model checking is deeply used in the industry now since it is one of the fully automatic techniques for programs verification. However, one drawback of this technique is the problem of state explosion especially when we compose complex systems together. Thus, model checking may lead to both time and space complexity. This is why, later more works using model checking are trying to solve this problem using *symbolic methods* [McM93] with more compact representations for the states and the atomic propositions (Binary Decision Diagram [Ran92]). Many tools that are used in industry are implementing this technology (for example SMV [smv], SPIN [spi]).

The SystemC waiting-state automata is perfectly used for model checking as mentioned in Chapter19.2, since the states space is substantially reduced, in fact, in the SystemC waiting-

state automaton model we consider only specific states where the component is visible and interacts with its environment. We apply predicate abstraction to reduce the control flow graph and we use a symbolic algorithm to reduce the composed automaton of the global system.

Abstract Interpretation Abstract interpretation [CC77, CC92] is a theoretical framework that provides definitions and criteria to simplify or abstract objects while ensuring that the abstraction is conform and accurate regarding a set of properties : if the property is not satisfied in the abstract object then it is not in the original object. It is based on the theory of fixpoints and sets that use approximations to reduce in a finite time calculations that may be potentially infinitely long. In computer science, abstract interpretation is based on tools that calculate in a finite time a sub-set of the program behavior, while the problem of computing an exact behavior of a program is undecidable [Ric53].

Verifying a property using the abstract description of the program may lead to three verdicts : *yes* the property is verified, *no* it is not, or *perhaps* my be because the abstraction is not that precise to make us decide. In the latter, we talk about a *false alarm* which means that the property is effectively verified in the program but it is not really verified in the abstraction because of the approximations. The issue of an abstract interpreter consists in finding the right balance between accuracy and the complexity of calculations on one hand, and the approximation and the speed of calculations on the other hand. Many areas of varying complexity have been developed to search a good compromise [e06]. Abstract interpretation is now successfully used in the industry with tools like Astrée [BBR03] capable to verify thousands of lines of an embedded critical code, but it focuses only on certain types of properties.

In this thesis, we intend to use abstract interpretation to especially improve the algorithms for the symbolic composition and reduction of the non-annotated automata (Chapter 8) and for automata annotated with parameters counters, the starting time and the duration (Chapter 9). We also intend to use abstract interpretation to analyze global systems written in SystemC and modeled with the SystemC waiting-state automata. This line of research allows us to verify more complex systems where the interactions between the parallel components are more complex and intricate and where constraints about functional and non-functional

behaviors are more strict.

Predicate abstraction which is a special variant of abstract interpretation is already a main step in our modeling process. We can say that the validation of the SystemC waiting-state automata is already proved during the generation of the automata from SystemC components.

List of Figures

List of Acronyms

A

ASM Abstract State Machine

AsmL Abstract State Machine Language

B

BCA Bus Cycle Accurate

BDD Binary Decision Diagram

C

CA Cycle Accurate

CEGAR Counter Example-Guided Abstraction Refinement

CFG Control Fow Graph

CMOS Complementary Metal-Oxide Semiconductor

CPU Control Processing Unit

D

DS Denotational Semantic

E

EDA Electronic Design Automation

F

FIFO First-In-First-Out

FSA Finite State Automaton

FSM Finite State Machine

H

HDLs Hardware Description Languages

HPIOM Heterogeneous Parallel Input/Output Machines

HW Hardware

I

ILP Integer Linear Programming

IPs Intellectual Property

ISA Instruction-Set Architecture

ISS Instruction Set Simulator

ITRS International Technology Roadmap for Semiconductor

L

LKS Labeled Kripke Structure

LTL Linear Temporal Logic

O

OO Object Oriented

P

PC Path Condition

PES Processing Elements

PV Programmer's View

PVT Programmer's View with Timing

Q

QoS Quality of Service

R

RT Register Transfer

RTL Register Transfer Level

S

SE Symbolic Execution

SoCs Systems-on-Chips

SOS Structural Operation Semantics

SW Software

T

TLM Transaction-Level Modeling

TL Transaction Level

TWSA Timed Waiting-State Automaton

U

UML Unified Modeling Language

V

VHDL Very High Speed Integrated Circuit

W

WCET Worst-Case Execution Time

WP Weakest Precondition

WSA Waiting State Automaton

Bibliography

[ACR10] I. Narasamdya A. Cimatti, A. Micheli and M. Roveri. Verifying systemc : a software model checking approach. In *in Formal Methods in Computer-Aided Design (FMCAD)*, 2010.

[AD94] R. ALUR and D. DILL. A theory of timed automata. *Theoretical Computer Science*, 126 (2) :183–235, 994.

[AG02] D.D. Gajski A. Gerstlauer. System-level abstraction semantics. Kyoto, Japan, October 2002.

[AGT04] A. Habibi A. Gawanmeh and S. Tahar. An executable operational semantics for systemc using abstract state machines. Technical Report 24, Technical Report, Concordia University, Department of Electrical and Computer Engineering, 2004.

[AHT06] H. Moinudeen A. Habibi and S. Tahar. Generating finite state machines from systemc. In *Georges G. E. Gielen, editor, DATE Designers' Forum*, pages 76–?81, Leuven, Belgium, 2006. European Design and Automation Association.

[AP05] A. Rybalchenko A. Podelski. Transition predicate abstraction and fair termination. In *In symposium on Principles of Programming Languages (POPL)*, 2005.

[Bal02] *The slam project : debugging system software via static analysis.* ACM, 2002.

[BBR03] R. Cousot J. Feret L. Mauborgne A. Miné D. Monniaux B. Blanchet, P. Cousot and X. Rival. A static analyzer for large safety-critical software. pages 196–207, San Diego, California, USA, June 2003. ACM Press.

[BBV08] B. Monsuez B. Benhamamouch and F. Védrine. Computing wcet using symbolic execution. In *Second International Workshop on Verification and Evaluation of Computer and Communication Systems*, Leeds, England, 2008.

[Ben11] B. Benhamamouch. *Calcul du pire temps d'exécution Méthode formelle s'adaptant á la sophistication croissante des architectures matérielles.* PhD thesis, Thèse de doctorat, UEI ENSTA-ParisTech et VERIMAG, 02 mai 2011.

[BM09] B. Benhamamouch and B. Monsuez. Computing worst case execution time (wcet) by symbolically executing a time-accurate hardware model. *International Journal of Design, Analysis and Tools for Circuits and Systems*, Vol.1(No.1), 2009.

[BS03] E. Borger and R.F. Stark. *Abstract State Machines : A Method for High-Level System Design and Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *In Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4) :511–547, August 1992.

[CFW99] M. Langenbach F. Martin M. Schmidt J. Schneider H. Theiling S. Thesing C. Ferdinand, D. Kastner and R. Wilhelm. Run-time guarantees for real-time systems ? the uses approach. *In GI Jahrestagung*, pages 410–419, 1999.

[CFW01] M. Langenbach F. Martin M. Schmidt H. Theiling S. Thesing R. C. Ferdinand, R. Heckmann and Wilhelm. Reliable and precise wcet determination for a real-life processor. In *In Proceedings of the International Conference on Embedded Software*, 2001.

[Cor08] A. Cortesi. Widening operators for abstract interpretation. pages 31–40, 2008.

[CP01] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *In Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, 2001.

[CS06] H. Cassé and P. Sainrat. Otawa, a framework for experimenting wcet computations. In *ERTS'06*, 2006.

[Dar88] J.A. Darringer. The application of program verification techniques to hardware verification. In *Annual ACM IEEE Design Automation Conference*, pages 376–381, 1988.

[De02] R. Drechsler and D. Große. Reachability analysis for formal verification of systemc. In *Euromicro Symposium on Digital Systems Design*, pages 337–340, 2002.

[De03] R. Drechsler and D. Große. Formal verification of ltl formulas for systemc designs. In *IEEE International Symposium on Circuits and Systems*, volume 25, pages 45–248, 2003.

[DGeD10] H. M. Le D. Groß e and R. Drechsler. Proving transaction and system-level properties of untimed systemc tlm designs. In *ACM & IEEE International Conference on Formal Methods and Models for Codesign*, 2010.

[DGZ00] R. Domer A. Gestlauer D. Gajski, J. Zhu and S. Zhao. *SpecC : Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[Dij97] E.W. Dijkstra. A discipline of programming. *Prentice Hall PTR*, 1997.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1990.

[D'S06] V. D'Silva. *Widening for automata*. PhD thesis, Institut fur Informatik, Universitaat Zurich, 2006.

[DTS08] G. Kamhi D. Tabakov, M. Vardi and E. Singerman. A temporal language for systemc. In *International Conf. on Formal Methods in CAD*, pages 1–9, 2008.

[e06] A. Miné. *The Octagon Abstract Domain*, volume 19. Higher-Order and Symbolic Computation, 2006.

[EC81] E. Edmund and M. Clarke. Design and synthesis of synchronization skeletons using branching time temporal logic. *In Logics of Programs*, 131, 1981. LNCS.

[EC04] M. Talupur D. Wang E. Clarke, O. Grumberg. High level verification of control intensive systems using predicate abstraction. In *In Proc. First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, volume 25. IEEE Computer Society, September 2004.

[ECP99a] I. Grumberg E. Clarke and D. Peled. Model checking. *The MIT Press*, 1999.

[ECP99b] O. Grumberg E. Clarke and D. Peled. *Model Checking*. MIT Press, 1999.

[ECS04] N. Sharygina J. Ouaknine E. Clarke, S. Chaki and N. Sinha. State/event-based software model checking. In *International Conf. on Integrated Formal Methods*, volume 25 of *105–127*, 2004.

[eD05a] D. Große and R. Dreschsler. Checksyc : An efficient property checker for rtl systemc designs. *In IEEE International Symposium on Circuits and Systems*, pages 4167–4170, 2005.

[ed05b] F. Ghenassia ed. Transaction-level modeling with systemc. *Springer*, 2005.

[FQ02] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *In proceedings, 29th Annual ACM Symposium on Principles and Programming Languages (POPL)*, pages 191–202, 2002.

[gap]

[GBR01] I. Jacobson G. Booch and J. Rumbaugh. Omg unified modeling language specification, September 2001.

[GK83] D. Gajski and R. Kuhn. *Guest editors introduction : New VLSI tools.* IEEE Computer, 1983.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *In Proc. of the 9th International Computer Aided Verification*, pages 72–83. Springer Verlag, 1997.

[Hav00] K. Havelund. Using runtime analysis to guide model checking of java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264, London, 2000. Springer-Verlag.

[HM09] N. Harrath and B. Monsuez. Timed systemc waiting-state automata. In *Third International Workshop on Verification and Evaluation of Computer and Communication Systems*, 2009.

[HM12] N. Harrath and B. Monsuez. *SystemC Waiting-State Automata*, volume Vol. 3 of *1/2*. On International Journal of Critical Computer-Based Systems (IJCCBS), 2012.

[HP00] K. Havelund and T. Pressburger. *Model Checking Java Programs Using Java PathFinder*, volume 2(4), chapter Software Tools for Technology Transfer (STTT), pages 366–381. 2000.

[HS06] T.A. Henzinger and J. Sifakis. The embedded system design challenge. In *In Proc. of the 14 International Symposium on Formall Methods.*, 2006.

[ITR] Design productivity gap. `http://public.itrs.net/files/1999_SIA_Roadmap/Design.pdf`, 1999 edition.

[JBU] F. Larsson P. Pettersson J. Bengtsson, K. G. Larsen and W. Yi. Uppaal. A tool suite for automatic verification of real-time systems. In *Workshop on Verification and Control of Hybrid Systems*. LNCS 1066, Springer.

[JLBP85] N. Halbwachs D. Pilaud J-L. Bergerand, P. Caspi and E. Pilaud. Outline of a real time data-flow language. In *In Real Time Systems Symposium, San Diego*, pages 33–42, September 1985.

[JTB98] B.O. Nnaji J.P. Terpenny and J.H. Bohn. Blending top-down and bottom-up approaches in conceptual design. *7th Annual Industrial Engineering Research Conference*, May 1998.

[K06] P. Karlsson, D. Eles and Peng Z. . Formal verification of systemc designs using a petri-net based representation. In *InProceeding on the conference on Design, Automation and Test in Europe*, pages 1228–1233, 2006.

[Kin76] J.C. King. *Symbolic execution and program testing.* 19(7). Communications of the ACM (Association for Computing Machinery), 1976.

[kle38] *On notation for ordinal numbers*, volume 3(4). 1938.

[KMS06] M. Mercaldi K.L. Man, A. Fedeli and M.P. Schellekens. *systemc*^{fl} : An infrastructure for a tlm formal verification proposal (with an overview on a tool set for practical formal verification of systemc descriptions). In *In Proceedings of the IEEE East-West Design & Test Workshop EWDTW*, 2006.

[KS05] D. Kroening and N. Sharygina. Formal verification of systemc by automatic hardware/software partitioning. In *the Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 101–110, 2005.

[LACP04] P. Eles L. A. Cortes and Z. Peng. Verification of embedded systems using a petri net based represenattion. In *In Proc. of the 13th International Symposium on System Synthesis*, pages 149–155, 2004.

[Lei05] S. Leibson. *The end of Moore's law.* Embedded Systems Design, December 2005.

[LL02] K.R.M. Leino and F. Logozzo. Loop invariants on demand. In *In K. Yi, editor, Proceedings, 3rd Asian Symposium on Programming languages and Systems (APLAS)*, volume 3780 of LNCS, pages 119–134, 2002.

[Lun02] T. Lundqvist. A wcet analysis method for pipelined microprocessors with cache memories. Goteborg, Sweeden, 2002.

[MA00] B. Marre and A. Arnould. Test sequences generation from lustre descriptions : Gatel. In *Proceedings of ASE-00 : The 15th IEEE Conference on Automated Software Engineering*, Grenoble, France, September 2000. IEEE CS Press.

[Man04] K.L. Man. *systemc^{FL}* : Formalization of systemc. In *In Proc of 12th IEEE Mediterranean Electrotechnical Conference*, 2004.

[MBW04] J. Klose B. Westphal M. Brill, W. Damm and H. Wittke. Live sequence charts : An introduction to lines, arrows, and strange boxes in the context of formal verification. *In SoftSpez Final Report*, 3147 of Lecture Notes in Computer Science :374–399, 2004. Springer.

[MCBG88] E. M. Clarke M. C. Browne and O. Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science - International Joint Conference on Theory and Practice of Software Development*, 59(1-2) :115–131, July 1988.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[MFM06] M. Moy and L. Maillet-Contoz F. Maraninchi. Lussy : an open tool for the analysis of systems-on-a-chip at the transaction level. 10 :26–35, 2006.

[MHP09] J.Hoenicke M. Heizmann and A. Podelski. Refinement of trace abstraction. *SAS*, pages 69–85, 2009.

[Moy05] M. Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transation Level.* PhD thesis, Phd thesis, Verimag laboratory Grenoble France, 2005.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[NB08] D. Kroening et N. Sharygina N. Blanc. Scoot : A tool for the analysis of systemc models. In *Proceedings of the Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, pages 467–470. Springer, tome 4963 de Lecture Notes in Computer Science, 2008.

[NH] B. Monsuez N. Harrath. *Compositional Reactive Semantics of System-Level Designs Written in SystemC and Formal Verification with Predicate Abstrac-*

tion. accepted in the International Journal of Critical Computer-Based Systems (IJCCBS), 2013.

[NH06] B. Niemann and Ch. Haubelt. Formalizing tlm with communicating state machines. In Forum of Specification and Design Languages, 2006.

[NHD11] B. Monsuez N. Harrath and J. Delacroix. Building systemc waiting state automata. In *Fifth International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2011)*, pages 54–66, 2011.

[PHG08] J. Fellmuth P. Herber and S. Glesner. Model checking systemc designs using timed automata. In *IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, 2008.

[Plo04] G. D. Plotkin. *A structural approach to operational semantics*, volume 60-61. Technical Report 19, University of Aahus, 1981. Also published in The Journal of Logic and Algebraic Programming, 2004.

[PMR05] B. Marre P. Mouy, N. Williams and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. *5th European Dependable Computing Conference*, pages 281–292, April 2005. Springer-Verlag.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46– ?57. IEEE Computer Society, 1977.

[pre] `http://www.predator-project.eu`.

[PXN06] H. Jifeng P. Xiaoqing, Z. Huibiao and J. Naiyong. An operational semantics of an event-driven system-level simulator. In *In proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 190–220, 2006.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *In Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[Ran92] E. B. Randal. *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams*, volume 24. ACM Computing Surveys, 1992.

[RAV93] T.A. Enzinger R. Alur and M. Y. Vardi. Parametric real-time reasoning. In *In the ACM Symposium on Theory of Computing*, 1993.

[Ric53] H. G. Rice. *Classes of Recursively Enumerable Sets and Their Decision Problems*, volume 74. Transactions of the American Mathematical Society, 1953.

[RSK07] R. Gupta R.K. Shyamasundar, F. Doucet and I.H. Kruger. Compositional reactive semantics of systemc and verification in rulebase. In *Proceedings of the 3rd international Haifa verification conference on Hardware and software : verification and testing HVC'07*, pages 34–50, Berlin, 2007. Springer-Verlag.

[Sal03] A. Salem. Formal semantics of synchronous systemc. In *Proc. Design, Automation and Test in Europe Conference and Exposition*, pages 10376–10381, 2003.

[SEE07] *Inferring Invariants by Symbolic Execution*, 2007.

[SL06] M. Kaufmann S. Leibson. Designing socs with configured cores. July 2006.

[smv] Le model checker smv. `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[SO03] A.Groce S.Chaki, E.Clarke and O.Strichman. *Abstraction with Minimum Predicates*. Springer Berlin/Heidelberg, 2003.

[spi] Le model checker spin. `http://spinroot.com/spin/whatispin.html`.

[SS88] David L. Dill S.Das and S.Park. Experience with predicate abstraction. In *In Computer Aided Verification (CAV)*, pages 160–171, 1988.

[Ste06] D. B. Stewart. Measuring execution time and real-time performance. In *Embedded Systems Conference*, Boston, September 2006.

[sys] Main page of the systemc initiative. `http://www.systemc.org.`

[sys02] *Open SystemC Initiative. SystemC v2.0 functional specification.* 2002.

[sys05] *Open SystemC Initiative. SystemC v2.1 language reference manual.* 2005.

[TA90] *Automata for modeling real-time systems.*, 1990.

[Tar55] A. Tarski. *A lattice-theoretical fixpoint theorem and its applications*, 1955.

[TBR01] T. Millstein T. Ball, R. Majumdar and K. Rajamani. Automatic predicate abstraction of c programs. In *In PLDI 2001*, 2001.

[THS09] P. Schachte T. Hansen and H. Sondergaard. State joining and splitting for the symbolic execution of binaries. *Notes in Computer Science*, Volume 5779/2009 :76–92, 2009.

[VC08] K. Lerman V. Crespi, A. Galstyan. *Top-down vs bottom-up methodologies in multi-agent system design*, volume 24(3). April 2008.

[Ven97] A. Venet. Abstract interpretation of the pi-calculus. *In Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, Springer, 1997.

[Ven98] A. Venet. Automatic determination of communication topologies in mobile systems. *In Proceedings of the Fifth International Symposium on Static Analysis (SAS'98)*, 1998.

[ver91] *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, 1991.

[vhd02] *IEEE Standard VHDL Language Reference Manual. IEEE Std 1076*. 2002.

[VPM11] N. Harrath V.A. Paun and B. Monsuez. A wcet estimation workflow based on the twsa model of systemc designs. *The 32nd IEEE Real-Time Systems Symposium*, November 2011.

[Wil04] R. Wilhelm. Why ai + ilp is good for wcet, but mc is not, nor ilp alone. *In Verification, Model Checking and Abstract Interpretation (VMCAI), LNCS 2937*, 2004.

[Win93] Glynn Winskel. The formal semantics of programming languages : an introduction. *MIT Press*, 1993.

[WMR03] J. Ruf W. Mueller and W. Rosenstiel. *SystemC Methodologies and Applications*. Kluwer Academic Publishers, 2003.

[WPdRP94] F. de Boer Y. Lakhneche Q. Xu W. P. de Roever, J. Hooman and P. Pandya. *State-Based Proof Theory of Concurrency : from noncompositional to compositional methods*. 1994.

[WRM01] D. Gerlach J. Kropf T. Rosenstiehl W. Ruf. Müller, J. Hoffmann. The simulation semantics of systemc. In *In Design, Automation and Test in Europe*. IEEE CS, 2001.

[YZM07] F. Védrine Y. Zhang and B. Monsuez. Systemc waiting-state automata. In *First International Workshop on Verification and Evaluation of Computer and Communication Systems*, pages 5–6, 2007.

[Zhu05] H. Zhu. *Linking the Semantics of a Multithreaded DiscreteEvent Simulation Language*. PhD thesis, London South Bank University, UK, 2005.

A Stepwise Compositional Approach to Model and Analyze SystemC Designs at the Transactional Level and the Delta Cycle Level

Nesrine Harrath

Laboratoire Informatique et Ingenierie des Systemes,

École Nationale de techniques Avancees (ENSTA ParisTech),

91762, Palaiseau Cedex.

Centre d' Études et de Recherche en Informatique et Communications (CEDRIC)

Conservatoire National des Arts et Métiers (CNAM),

75141, Paris Cédex 03.

Résumé : Les systèmes embarqués sont de plus en plus intégrés dans les applications temps réel actuelles. Ils sont généralement constitués de composants matériels et logiciels profondément intégrés mais hétérogènes. Ces composants sont développés sous des contraintes très strictes. En conséquence, le travail des ingénieurs de conception est devenu plus difficile. Pour répondre aux normes de haute qualité dans les systèmes embarqués de nos jours et pour satisfaire aux besoins quotidiens de l'industrie, l'automatisation du processus de développement de ces systèmes prend de plus en plus d'ampleur. Un défi majeur est de développer une approche automatisée qui peut être utilisée pour la vérification intégrée et la validation de systèmes complexes et hétérogènes.

Dans le cadre de cette thèse, nous proposons une nouvelle approche compositionnelle pour la modélisation et la vérification des systèmes complexes décrits en langage SystemC. Cette approche est basée sur le modèle des SystemC Waiting State Automata (WSA). Les SystemC Waiting State Automata sont des automates permettant de modéliser le comportement abstrait des systèmes matériels et logiciels décrits en SystemC tout en préservant la sémantique de l'ordonnanceur SystemC au niveau des cycles temporelles et au niveau des delta-cycles. Ce modèle permet de réduire la complexité de la modélisation des systèmes complexes due au problème de l'explosion combinatoire tout en restant fidèle au système initial. Ce modèle est compositionnel et supporte le raffinement. De plus, il est étendu par des paramètres temps ainsi que des compteurs afin de prendre en compte les aspects relatifs à la temporalité et aux propriétés fonctionnelles comme notamment la qualité de service.
Dans le cadre de cette thèse, nous proposons une chaîne de construction automatique des WSAs à partir de la description SystemC. Cette construction repose sur l'exécution symbolique et l'abstraction des prédicats. Nous proposons un ensemble d'algorithmes de composition et de réduction de ces automates afin de pouvoir étudier, analyser et vérifier les comportements concurrents des systèmes décrits ainsi que les échanges de données entre les différents composants. Nous proposons enfin d'appliquer notre approche dans le cadre de la modélisation et la simulation des systèmes complexes. Ensuite l'expérimenter pour donner une estimation du pire temps d'exécution (worst-case execution time (WCET)) en utilisant le modèle du Timed SystemC WSA. Enfin, on définit l'application des techniques du model checking pour prouver la correction de l'analyse abstraite de notre approche.
Mots clés : SystemC, Méthodes Formelles, Automates, Exécution Symbolique, Abstraction des prédicats, Sémantiques des langages, Model Checking.

Abstract : Embedded systems are increasingly integrated into existing real-time applications. They are usually composed of deeply integrated but heterogeneous hardware and software components. These components are developed under strict constraints. Accordingly, the work of design engineers became more tricky and challenging. To meet the high quality standards in nowadays embedded systems and to satisfy the rising industrial demands, the automatization of the developing process of those systems is gaining more and more importance. A major challenge is to develop an automated approach that can be used for the integrated verification and validation of complex and heterogeneous HW/SW systems.

In this thesis, we propose a new compositional approach to model and verify hardware and software written in SystemC language. This approach is based on the SystemC Waiting State Automata (WSA). The SystemC Waiting State Automata are used to model the abstract behavior of hardware or software systems described in SystemC, they preserve the semantics of the SystemC scheduler at the temporal and the delta-cycle level. This model allows to reduce the complexity of the modeling process of complex systems due to the problem of state explosion during modeling while remaining faithful to the original system. The SystemC waiting state automaton is also compositional and supports refinement. In addition, this model is extended with parameters such as time and counters in order to take into account further aspects like temporality and other extra-functional properties such as QoS.
In this thesis, we propose a stepwise approach on how to automatically extract the SystemC WSAs from SystemC descriptions. This construction is based on symbolic execution together with predicate abstraction. We propose a set of algorithms to symbolically compose and reduce the SystemC WSAs in order to study, analyze and verify concurrent behavior of systems as well as the data exchange between various components. We then propose to use the SystemC WSA to model and simulate hardware and software systems, and to compute the worst cas execution time (WCET) using the Timed SystemC WSA. Finally, we define how to apply model checking techniques to prove the correctness of the abstract analysis.
Keywords : SystemC, Formal Methods, Automata, Symbolic Execution, Predicate Abstraction, Semantics of Programming Languages, Model Checking.