

N° attribué par la bibliothèque

|---|---|---|---|---|---|---|---|---|

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : « INFORMATIQUE : SYSTÈMES ET COMMUNICATION »

préparée au laboratoire VERIMAG

dans le cadre de l'École doctorale « **MATHÉMATIQUES, SCIENCES ET
TECHNOLOGIES DE L'INFORMATION, INFORMATIQUE** »

présentée et soutenue publiquement

par

Claude HELMSTETTER

le 26 mars 2007

Titre :

**Validation de modèles de systèmes sur puce en
présence d'ordonnancements indéterministes et
de temps imprécis**

Directrice de thèse :

Florence Maraninchi

JURY

Marc Renaudin
Gérard Berry
Patrice Godefroid
Florence Maraninchi
Laurent Maillet-Contoz

Président
Rapporteur
Rapporteur
Directrice de thèse
Examineur

Remerciements

(page planifiée pour le jour de la soutenance)

Table des matières

1	Introduction	9
1.1	L'essor des modèles transactionnels pour la conception des SoCs	9
1.2	La validation au niveau transactionnel	10
1.3	Le problème de l'ordonnancement et des synchronisations	11
1.4	Notre solution pour le problème de l'ordonnancement	12
1.5	Nos autres réalisations	13
1.5.1	Validation de modèles TLM en présence de temps imprécis	13
1.5.2	Cadre formel pour la parallélisation du simulateur SystemC	14
1.6	Contenu du document	15
1.6.1	Résumé des contributions	15
1.6.2	Plan du document	16
2	Modélisation des systèmes sur puce	19
2.1	Le flot de conception des systèmes sur puce	20
2.1.1	Partitionnement logiciel - matériel	20
2.1.2	Les différents niveaux d'abstraction	21
2.1.3	Validation : vérification, simulation et test	23
2.2	Les modèles transactionnels	25
2.2.1	Concepts communs	25
2.2.2	Les modèles fonctionnels (PV)	27
2.2.3	Les modèles temporisés (PVT)	27
2.3	SystemC et la librairie TLM	28
2.3.1	SystemC	28
2.3.2	La librairie TLM	30
3	Validation des modèles transactionnels	35
3.1	Vérification formelle	35
3.1.1	Outils candidats pour la validation des modèles de SoCs	36
3.1.2	La chaîne d'outil LUSSY	36
3.2	Validation par simulations	37
3.2.1	Environnement de tests	37
3.2.2	Outils pour la génération de tests	39
3.3	Combinaison des méthodes	40
3.3.1	Génération de tests ciblant un trou de couverture	41
3.3.2	Vérification à la volée et simulations équivalentes	42

4	Le problème de l'ordonnement	43
4.1	L'ordonnement en SystemC	44
4.1.1	Algorithme de l'ordonneur	44
4.1.2	Les actions de communication	46
4.1.3	Ajout : l'instruction "yield"	46
4.2	Exemple	47
4.2.1	Exemple avec deux processus	47
4.2.2	Version étendue à trois processus	48
4.3	Conséquences pour des cas réels	50
4.3.1	Blocage au démarrage	50
4.3.2	Procédure d'arbitrage	50
4.4	Réalisation de systèmes indépendants de l'ordonnement	52
4.4.1	Exemple : système d'arbitrage indépendant de l'ordonnement	52
4.4.2	Analyse de l'exemple et limitations	53
4.4.3	Conclusion	54
5	Génération automatique d'ordonnements	55
5.1	Introduction	55
5.1.1	Objectif	55
5.1.2	Principe général	56
5.1.3	Contenu et plan du chapitre	56
5.2	Séparation de l'ordonnement et des données	57
5.2.1	Données fixées statiquement	57
5.2.2	Données générées dynamiquement	57
5.3	Représentation formelle	58
5.3.1	Système sous-test et ordonnements	58
5.3.2	Relations entre les ordonnements	59
5.3.3	Représentations graphiques	62
5.3.4	Égalité de transitions et lien avec le code source du SSTD	64
5.4	Algorithmes	66
5.4.1	Relation de commutativité	66
5.4.2	Ordre causal et permutabilité	70
5.4.3	Génération des nouveaux ordonnements	71
5.5	Mise en application pour la validation	75
5.5.1	Propriété principale	75
5.5.2	Conséquences pour la validation	78
6	Implantation	81
6.1	Architecture	82
6.2	Ordonneur interactif	83
6.2.1	Interface avec l'implantation OSCI	83
6.2.2	Fonctionnalités du nouvel ordonneur	84
6.3	Enregistrement des traces d'exécutions	85
6.3.1	Contenu et format	85
6.3.2	Modification du noyau SystemC	87
6.3.3	Instrumentation du modèle	87
6.4	Calcul des dépendances pour une exécution	92

6.4.1	Analyse syntaxique du fichier XML	92
6.4.2	Les structures de données	93
6.4.3	Calcul de l'ordre partiel	93
6.5	Génération de l'ensemble des ordonnancements	96
6.6	Outils annexes	97
6.6.1	Génération des graphiques de dépendances	97
6.6.2	Enregistrement d'une trace détaillée	99
6.6.3	Arbres des contraintes d'ordonnement	101
6.7	Conclusion	101
7	Evaluation et étude de cas	103
7.1	Les cas élémentaires	104
7.1.1	Exemple avec impasse possible	104
7.1.2	Exemple avec génération d'ordonnements équivalents	105
7.2	Test de performance	107
7.2.1	Indexeur	107
7.2.2	Modèle TLM dédié à des travaux pratiques	109
7.3	Cas réel	110
7.3.1	Description	110
7.3.2	Instrumentation du modèle	111
7.3.3	Validation et dépouillement des résultats	112
7.3.4	Prise en compte des événements persistants	114
7.3.5	Tentative avec la plateforme complète	116
7.4	Bilan	117
8	Génération de schémas de temporisation	121
8.1	Contexte et motivations	122
8.1.1	Système plongé dans un environnement non-déterministe	122
8.1.2	Système partiellement temporisé	123
8.1.3	Exemple introductif	124
8.1.4	Méthodes existantes pour le choix des délais effectifs	125
8.2	Principe	126
8.2.1	Séparation de l'ordonnement, des durées et des données	126
8.2.2	Les modèles TLM que nous considérons	126
8.2.3	Aperçu général	128
8.2.4	Algorithme formel	130
8.3	Implantation	134
8.3.1	Construction des contraintes linéaires	134
8.3.2	Résolution des systèmes linéaires	135
8.3.3	Représentation des schémas de temporisation	135
8.4	Évaluation	136
8.5	Autres approches pour la validation de modèles avec temps imprécis	136
8.5.1	Test, plus réduction d'ordre partiel ou programmation linéaire	136
8.5.2	Extraction d'un modèle puis vérification formelle	137

9	Vers un simulateur SystemC parallèle pour modèles TLM	139
9.1	Objectif et contraintes	139
9.1.1	Le modèle d'exécution actuel de l'OSCI	140
9.1.2	Parallélisation : aperçu général	141
9.1.3	Parallélisation : respect de la spécification	141
9.2	Cadre formel : dépendances pour la parallélisation	143
9.2.1	Définitions : transitions, actions et exécutions parallèles	143
9.2.2	Relation d'indépendance pour la parallélisation	143
9.3	Outils existants, approche structurelle	145
9.3.1	Les modèles SystemC avec communications globalement synchrones	145
9.3.2	Outils existants	146
9.3.3	Relâchement de la non-préemptivité dans le cas des transactions	147
9.4	Vers un outil adapté au TLM, approche non-structurelle	148
9.4.1	Les granularités envisageables	148
9.4.2	Sketch d'ordonnanceur multiprocesseur	150
9.4.3	Analyses statiques pour le pré-calcul des dépendances	151
9.4.4	Identification dynamique des transitions	152
9.5	Perspectives	152
10	Conclusion et perspectives	153
10.1	Bilan	153
10.1.1	Mise en évidence du problème de l'ordonnement	154
10.1.2	Détection des erreurs de synchronisation	154
10.1.3	Extension aux modèles avec temps imprécis	155
10.1.4	Autres contributions	155
10.2	Perspectives	156
10.2.1	Amélioration des performances	156
10.2.2	Réduction du nombre d'entrelacements visités pour LUSY	158
10.2.3	Extensions vers d'autres espaces de données ou d'autres contextes	159
	Annexes	159
A	Classe pour un arbitrage indépendant de l'ordonnement	161
A.1	Entête	161
A.2	Implantation	162
A.3	Test	163
B	Test de performance : l'indexeur	165
B.1	Code source complet	165
B.2	Comparaison avec la version instrumentée	167
	Bibliographie	167
	Index	177

Chapitre 1

Introduction

Sommaire

1.1	L’essor des modèles transactionnels pour la conception des SoCs	9
1.2	La validation au niveau transactionnel	10
1.3	Le problème de l’ordonnancement et des synchronisations	11
1.4	Notre solution pour le problème de l’ordonnancement	12
1.5	Nos autres réalisations	13
1.5.1	Validation de modèles TLM en présence de temps imprécis	13
1.5.2	Cadre formel pour la parallélisation du simulateur SystemC	14
1.6	Contenu du document	15
1.6.1	Résumé des contributions	15
1.6.2	Plan du document	16

1.1 L’essor des modèles transactionnels pour la conception des SoCs

Les appareils informatiques de notre vie quotidienne sont soumis à des contraintes fortes : ils doivent par exemple fonctionner à des vitesses élevées pour les traitements multimédia, ou consommer peu pour allonger l’autonomie des systèmes embarqués. Par ailleurs, leur coût pour l’utilisateur final doit resté maîtrisé. Pour cela, il faut faire tenir une grande puissance de calcul sur une petite surface. Le principe des *systèmes sur puce* (SoC) consiste à regrouper sur un même circuit intégré tous les composants informatiques d’un appareil, par exemple des mémoires, des processeurs ou des convertisseurs analogiques. L’hétérogénéité entre les parts analogiques, numériques et logicielles de ces puces rend leur conception longue et complexe. Permettre aux SoCs de suivre l’évolution rapide des besoins nécessite des efforts dans plusieurs domaines, dont la physique, l’informatique, et même la finance car le coût des chaînes de production semble aussi suivre la loi de Moore, obligeant à des unions entre grands groupes.

Le cœur du flot de conception des SoCs est le niveau “*transfert de registres*” (RTL). Les descriptions RTL sont très précises : la valeur de chaque bit d’information est connue à chaque top d’horloge. Des outils automatiques permettent ensuite d’obtenir le *layout*, c’est-à-dire la disposition des portes logiques, qui permet la fabrication physique de la puce. L’obtention de la puce physique à partir d’une description s’appelle la *synthèse*. Cependant, la simulation des descriptions RTL est trop lente pour permettre le développement et la validation du logiciel embarqué. Il faut environ 1 heure pour le

décodage et encodage d'une image MPEG4. De plus, les descriptions RTL sont disponibles trop tard, par rapport à la date de mise sur le marché. Pour cette raison, les descriptions RTL ne conviennent ni pour le développement du logiciel embarqué, ni pour l'évaluation des choix d'architecture qui doivent être effectués tôt.

La solution, actuellement en plein essor, consiste à développer des modèles complémentaires, qui abstraient toutes les informations non primordiales. Ce nouveau niveau d'abstraction est appelé *niveau de modélisation transactionnel* (TLM). Les modèles TLM sont exécutables et offrent de très bonnes vitesses de simulation (environ 3 secondes pour le décodage d'une image MPEG4). De plus, ils peuvent être disponibles très tôt car il sont plus rapides à écrire que le RTL. Il n'existe actuellement aucun outil capable de construire automatiquement une description RTL à partir d'un modèle transactionnel (les modèles TLM ne contiennent pas assez d'information pour cela). Le niveau transactionnel est lui-même divisé en sous-niveaux. Le développement du logiciel embarqué se fait sur les *modèles fonctionnels* qui sont les plus abstraits, et qui offrent donc la meilleure vitesse de simulation. Les évaluations de performances se font sur des *modèles transactionnels temporisés*, qui contiennent des informations supplémentaires sur la micro-architecture, mais qui sont encore beaucoup plus abstraits que le RTL.

Aucun des langages prévus pour le matériel (VHDL, Verilog, ...), ou pour le logiciel (C++, Java, ...), ne permet en l'état un développement efficace des modèles de SoCs. Une collaboration entre plusieurs industriels, l'OSCI pour *Open SystemC Initiative*, a mis au point un nouveau langage : SystemC, conçu comme une extension de C++. Celui-ci vise et parvient à rassembler les avantages de la programmation logicielle et de la description matériel. SystemC permet de décrire l'architecture du SoC : les composants matériels sont représentés par des *modules* qui sont reliés via des *ports* et des *canaux* de communication. Le comportement du matériel et du logiciel embarqué est décrit grâce à des *processus* exécutant du code C++ général. Tous les composants peuvent ainsi se modéliser en SystemC, quelle que soit leur nature, et à tous les niveaux d'abstraction du flot de conception des SoCs. Plusieurs industries ont ensuite développé chacune de son côté des bibliothèques pour les communications au niveau d'abstraction transactionnel. L'OSCI travaille actuellement à leur uniformisation, via l'élaboration et la normalisation d'une bibliothèque pour modèles transactionnels, proche de celle déjà conçue et utilisée par STMicroelectronics.

L'équipe SPG (*System Platform Group*) de STMicroelectronics, est en charge du développement de la bibliothèque TLM et des outils associés. En 2002, elle a entamé une collaboration avec l'équipe *Synchrone* du laboratoire *Verimag*, afin de travailler avec les membres de la recherche publique sur des problèmes dépassant les compétences internes de l'entreprise. L'objectif de cette collaboration, fixé par l'industrie, est de développer des méthodes et outils pour l'obtention de modèles transactionnels fiables et robustes. Pour cela, il est possible à la fois d'améliorer les méthodes de conception des modèles, et de développer des techniques de validation adaptées. Les travaux de thèse décrits dans ce document s'inscrivent plus précisément dans le cadre de la validation par simulations de modèles SystemC-TLM.

1.2 La validation au niveau transactionnel

Étant donné un modèle transactionnel, la première étape est de s'assurer qu'il ne contient pas d'erreurs intrinsèques. Il s'agit de vérifier des propriétés génériques, par exemple l'absence d'interblocage (*deadlock*), et des propriétés issues des spécifications informelles. La première réalisation issue de la collaboration ST-Verimag a été la chaîne d'outil LusSy, développée par Matthieu Moy. Cette chaîne

d'outils permet d'extraire des modèles formels à partir de modèles SystemC-TLM, qui sont ensuite fournis à des outils de vérification formelle. Cela permet d'ores et déjà de prouver des propriétés sur des systèmes de petites tailles. Des améliorations sont prévues : la description de l'architecture présente dans les modèles TLM peut par exemple être exploitée pour de la vérification compositionnelle.

Les modèles TLM fonctionnels servent de référence pour la validation du logiciel embarqué. Il est donc important que le logiciel s'exécute de la même façon sur le modèle et sur le vrai système. Notamment, il faut s'assurer que toute propriété fonctionnelle déjà prouvée sur le modèle abstrait soit encore correcte sur le matériel. Il s'agit d'un problème important, qui est traité au sein de la coopération ST-Verimag. L'idée est de traiter le problème en deux étapes, via l'écriture de modèles temporisés corrects par construction (Jérôme Cornet), puis par comparaison de ces modèles intermédiaires avec le RTL (Giovanni Funchal).

Une troisième tâche consiste à valider le logiciel embarqué lui-même. Pour cela, il est nécessaire de considérer l'ensemble du SoC, matériel et logiciel inclus. Cela donne des systèmes complexes et hétérogènes, car contenant par exemple : la description de l'architecture, du code pour gérer les synchronisations (e.g. arbitrage dans le contrôleur d'interruption), des algorithmes de référence pour les traitements multimédia (e.g. décodage MPEG), plus le logiciel embarqué qui peut contenir un système d'exploitation épuré. La *validation par simulations* semble ici la solution la plus adéquate.

La validation par simulations consiste à générer des entrées pour le système sous test, à l'exécuter avec ces entrées, puis à contrôler que ses sorties sont conformes à la spécification. Des notions de couverture permettent d'estimer l'avancée de la validation et de diriger les prochains tests. Les simulations ne permettent pas de prouver formellement des propriétés. Cependant, les mesures de couverture permettent de fournir des garanties suffisantes par rapport à nos besoins (e.g. applications multimédia). De plus, les erreurs logicielles sont plus simples et moins coûteuses à corriger que les erreurs matérielles, lorsqu'elles sont détectées tardivement (une erreur dans le RTL peut obliger à une reprise de masque : une dépense de l'ordre du million d'euros).

Pour les modèles TLM, les tests consistent généralement en du logiciel embarqué qui est exécuté par des modèles de processeur, et des instructions pour les bouchons modélisant les entrées et sorties du système. Nous devons valider le comportement de tests écrits spécifiquement pour valider un composant, ainsi que le vrai logiciel qui sera embarqué sur le SoC final. Jusqu'à présent, l'écriture de ces tests se fait sans générateur automatique, notamment à cause du manque de spécifications formelles qui pourraient servir d'entrée à ce type d'outils.

1.3 Le problème de l'ordonnancement et des synchronisations

Les systèmes matériels sont essentiellement parallèles car composés de nombreuses portes logiques calculant simultanément. Même en regardant un SoC avec plus de recul, plusieurs entités parallèles sont encore identifiables : processeurs multicœurs, DMA, arbitre de bus, mémoires, etc. Or, nous simulons ces systèmes matériels via du logiciel qui s'exécute sur un seul processeur. Même si l'on dispose d'un simulateur fonctionnant sur une machine multiprocesseur, le nombre de processeurs du simulateur reste bien inférieur au nombre d'entités parallèles du système physique. Cela oblige à *ordonner* l'exécution des entités parallèles du SoC, qui sont représentées en SystemC par des processus.

Les modèles TLM fonctionnels ne contiennent pas d'horloge ; ils sont essentiellement asynchrones. Cela représente bien le parallélisme de systèmes physiques dont la temporisation exacte est encore inconnue. En effet, l'ordre des interactions entre entités parallèles du vrai système n'est

connu qu'à la sortie de la description RTL, car cet ordre dépend des détails de micro-architecture. Pour profiter de la sémantique asynchrone des modèles TLM, il est nécessaire d'envisager plusieurs ordonnancements différents.

La spécification du langage SystemC définit une sémantique indéterministe pour l'ordonnanceur. C'est une bonne chose puisque cela nous permet de représenter plus fidèlement le matériel. Cependant, l'implantation fournie par l'OSCI, et qui est majoritairement utilisée dans l'industrie, est déterministe. Une simulation avec juste cet implantation risque de ne pas être représentative du système matériel final, puisqu'elle ne peut explorer qu'un ordre parmi tous les ordres possibles sur le matériel. Le comportement observé peut dépendre de l'ordonnancement, même si les données sont fixées. Cela est généralement volontaire car motivé par des soucis de représentativité du vrai parallélisme du matériel. Cependant, une dépendance à l'ordonnancement peut aussi être involontaire et mener à un comportement erroné. Cela est un problème pour la validation par simulations : il faut *couvrir* l'espace des ordonnancements en plus de celui des données.

Le cœur d'un ordonnanceur peut être vu comme une fonction qui reçoit une liste de processus éligibles et renvoie son choix parmi cette liste. Une solution imaginable consiste à implanter cette fonction avec un générateur aléatoire. Ainsi, il est possible d'observer plus de comportements. Cependant, cela ne permet pas de maîtriser la couverture. Trouver des solutions pour la mesurer n'est pas suffisant ; il faut aussi une solution pour compléter les trous de couverture.

Notons que ce problème de couverture n'apparaît pas avec les méthodes de vérification formelle, puisqu'elles travaillent sur des modèles qui contiennent tous les choix possibles. Cela augmente la taille de l'espace d'états à explorer, mais des techniques de réduction d'ordre partiel, ou de vérification symbolique, permettent de limiter l'effet de l'explosion du nombre d'états.

Comment faire communiquer des composants dans un modèle transactionnel est désormais parfaitement expliqué lors des formations qui s'adressent aux futurs développeurs. En revanche, le problème des dépendances à l'ordonnancement et la programmation des synchronisations entre processus furent longtemps laissés en exercice. Celui-ci consistait à développer un arbitre avec priorités pour un modèle de bus, grâce aux briques rudimentaires fournies par le langage SystemC. Au commencement de cette thèse, nous avons rapidement identifié des manquements dans la solution finalement donnée à l'exercice ci-dessus : son comportement dépendait plus de l'ordonnancement que prévu. Les mauvaises synchronisations peuvent avoir divers effets néfastes : inter-blocage dû à des attentes simultanées et réciproques, corruption de données due à un entrelacement incorrect des accès, inversion de priorités, etc. Notre tâche est d'éviter que de telles erreurs de synchronisation se retrouvent dans les versions finales des modèles industriels.

Dans cette thèse, nous traitons l'indéterminisme de l'ordonnancement en SystemC, mais le même problème se pose avec tous les langages de modélisation du matériel, et plus généralement de systèmes avec du vrai parallélisme.

1.4 Notre solution pour le problème de l'ordonnancement

En simulant un test avec un seul ordonnancement, seul un comportement parmi d'autres est observé. Plusieurs simulations avec un ordonnanceur aléatoire montrent plus de comportements, mais cela ne fournit aucune garantie d'avoir couvert tous les comportements possibles. Le nombre total d'ordonnements possibles pour un test réel est bien trop grand pour qu'un parcours exhaustif soit possible.

Notre solution repose sur le fait qu'exécuter tous les ordonnancements possibles n'est pas

nécessaire pour trouver tous les comportements erronés. Grâce à des connaissances sur les synchronisations entre processus, nous pouvons déduire que deux ordonnancements sont équivalents, c'est-à-dire suffisamment semblables pour que la présence d'une erreur avec l'un soit déductible de la simulation avec l'autre. C'est sur ce principe que sont basées les techniques de réductions d'ordre partiel. L'une de ces techniques, connue sous le nom de *réduction d'ordre partiel dynamique*, a toutes les qualités requises pour être utilisées dans le cadre de la validation par simulations.

Nous avons décidé d'appliquer cette technique pour couvrir l'espace des ordonnancements. Le principe général est le suivant : nous commençons par exécuter le test, tel que défini ci-dessus, avec un ordonnancement quelconque. Chaque fois que nous suspectons que des choix d'ordonnancement mènent à des comportements différents, nous ré-exécutons ce test avec un nouvel ordonnancement. L'idée consiste à observer les actions effectuées par chaque processus afin de deviner si un ordonnancement différent aurait pu mener à un résultat différent. Pour décider si un nouvel ordonnancement est nécessaire, nous utilisons un critère approximatif dans le sens suivant : nous pouvons engendrer plusieurs ordonnancements menant au même résultat, mais nous ne pouvons considérer comme équivalents deux ordonnancements qui mènent à des résultats différents. Le résultat final est un jeu d'ordonnements suffisamment riche pour offrir les garanties nécessaires pour la validation, et suffisamment concis pour être entièrement exécuté, y compris dans le cas de programmes réels. Concrètement, nous générons tous les états finaux possibles, ce qui permet de détecter toutes les erreurs locales, ainsi que tous les inter-blocages.

Cette technique de réduction d'ordre partiel étant très récente (publiée en Janvier 2005), il s'agit de l'une des premières applications. Certaines adaptations sont nécessaires pour pouvoir traiter les modèles SystemC-TLM. Notamment, il faut prendre en compte la non-préemptivité de SystemC, et les structures de synchronisations spécifiques. Par ailleurs, nous avons modifié l'algorithme principal, qui repose ainsi sur le parcours d'un *arbre de contraintes d'ordonnements*, plutôt que sur celui d'un *arbre d'ordonnements réduit*.

L'implantation de cette technique constitue le cœur de notre chaîne d'outil. Pour obtenir des traces d'exécution suffisamment précises, une étape préalable d'instrumentation du code source est nécessaire. L'objectif est de détecter les accès aux variables partagées, c'est-à-dire accessibles par au moins deux processus SystemC distincts. Nous ne disposons pas encore d'un outil automatique et complet, mais les solutions déjà en place permettent d'accomplir cette tâche manuellement pour un coût acceptable. Les sections de code qui n'accèdent pas directement à des variables partagées, n'ont pas besoin d'être instrumentées. Cela permet de valider du logiciel embarqué qui respecte ce critère et qui n'est disponible que sous forme binaire.

Enfin, nous avons développé un ensemble d'outils auxiliaires destinés à faciliter la compréhension des synchronisations, et la localisation des erreurs détectées. Ces outils ont rencontré beaucoup d'intérêts dans l'équipe SPG de STMicroelectronics.

1.5 Nos autres réalisations

Le problème de l'ordonnement fut notre principale préoccupation, mais nous nous sommes aussi intéressés aux deux problèmes ci-dessous.

1.5.1 Validation de modèles TLM en présence de temps imprécis

L'indéterminisme de l'ordonneur permet de modéliser un ensemble de comportements plutôt qu'un seul. Idéalement, les modèles fonctionnels ne devraient contenir aucune information tempo-

relle. Dans ce cas, l'ensemble des ordonnancements possibles représente un large sur-ensemble des comportements du matériel. Cependant, les ingénieurs sont souvent contraints d'ajouter des annotations temporelles pour éviter des comportements irréalistes. Par exemple, le temps entre l'affichage de deux images ($1/25^e$ de seconde) ne doit pas être plus court que le délai nécessaire pour une lecture d'un octet en mémoire. Si ces annotations temporelles sont codées sous la forme de durées fixes, l'ordonnanceur SystemC les traduit par des synchronisations globales beaucoup trop fortes. Dans ce cas, l'ensemble des ordonnancements possibles devient vite très petit et ne constitue plus qu'un sous-ensemble des comportements réalistes.

La solution, choisie en collaboration avec les ingénieurs de STMicroelectronics, consiste à utiliser des annotations imprécises, concrètement des intervalles représentant l'ensemble des durées réalistes. Ces annotations sont intégrées au modèle grâce à une nouvelle instruction `PV_wait(durée D, marge d)`, qui signifie que le processus contenant cette instruction doit attendre une durée comprise entre $D-d$ et $D+d$. En faisant varier la largeur de chaque intervalle, il est possible d'obtenir un sur-ensemble d'exécutions proche de celui des comportements réalistes.

Pour simuler un modèle possédant des annotations temporelles imprécises, il est nécessaire de choisir une durée effective chaque fois que nous rencontrons l'une de ces instructions. Cela soulève le même problème que pour l'indéterminisme de l'ordonnancement : une simple exécution d'un test avec des durées arbitraires n'est pas assez représentative, plusieurs exécutions avec des durées aléatoires ne fournissent aucune garantie, et enfin l'espace des jeux de durées valides est infini. Dans le cadre de la vérification formelle, ces modèles transactionnels avec temps imprécis pourraient se modéliser par des *automates temporisés*, ce qui permettrait de traiter symboliquement toutes les durées autorisées.

Notre solution se présente comme une extension de celle mise en œuvre pour le problème de l'ordonnancement. Elle repose sur le même principe fondamental, à savoir que les jeux de durées peuvent se regrouper en classes d'équivalence, telles qu'il soit suffisant d'exécuter un seul représentant de chaque classe pour trouver toutes les erreurs.

L'algorithme général reste le même : nous exécutons le test avec des durées et un ordonnancement quelconques, puis nous examinons en détail les communications qui ont eu lieu pour générer de nouvelles valeurs susceptibles de mener à un état final différent. L'analyse des synchronisations nous donne une liste de permutations à envisager dans l'ordonnancement courant. Ces propositions de permutations sont ensuite codées sous la forme de systèmes de contraintes linéaires, qui représentent les contraintes temporelles issues des annotations. Si un de ces systèmes n'a pas de solution, alors la permutation correspondante est impossible. Dans le cas contraire, il suffit de choisir l'une des solutions, et de l'utiliser comme un jeu de durées pour une nouvelle exécution. Nous recommençons ensuite itérativement sur chaque nouvelle exécution pour générer de proche en proche tout un ensemble de jeux de durées. Chaque jeu de durées généré est accompagné d'un ordonnancement, ou plus exactement de contraintes permettant la génération d'un ordonnancement.

Ce nouvel algorithme fournit la même garantie que celui limité à l'ordonnancement : nous détectons toutes les erreurs locales qui peuvent survenir pour un test donné. Nous verrons que cet algorithme passe moins bien à l'échelle mais est tout de même applicable à des systèmes de taille moyenne.

1.5.2 Cadre formel pour la parallélisation du simulateur SystemC

En 2006, Yussef Bouzouzou s'est attaqué à un nouveau problème, dans le cadre d'une coopération étendue Silicomp - Verimag - STMicroelectronics. L'objectif est d'accélérer les simulations en développant un nouveau simulateur SystemC qui profite des machines multiprocesseurs, tout en restant conforme à la spécification SystemC. Lors de cette thèse, nous avons participé à ce projet et y

avons apporté des contributions théoriques.

Une solution naïve consiste à associer chaque processus SystemC à un processus du système d'exploitation. Cela n'est ni efficace, ni correct : premièrement, le système d'exploitation se retrouve surchargé, et deuxièmement, la non-préemptivité de SystemC est violée.

Pour respecter la non-préemptivité, il est nécessaire d'interdire certaines parallélisations en fonction des dépendances entre transitions de processus concurrents, une transition étant ici une section de code atomique pour l'ordonnanceur SystemC. Ces dépendances sont semblables à celles considérées par notre outil de génération automatique d'ordonnancements, mais nécessitent cette fois une analyse statique. Plusieurs outils, basés sur une approche structurelle, ont été proposés et développés (dont l'un au sein de l'équipe SPG). Cette approche consiste à considérer que deux transitions éligibles à un même instant sont indépendantes dès qu'elles appartiennent à des composants différents.

Nous montrons dans ce document que l'approche structurelle n'est pas adaptée aux modèles TLM fonctionnels. Nous donnons finalement les principes de bases pour la réalisation d'un nouveau simulateur SystemC pour machines multiprocesseurs, à la fois conforme à la spécification, et plus efficace car permettant plus de parallélisations.

1.6 Contenu du document

1.6.1 Résumé des contributions

Le principal résultat pratique de cette thèse consiste en le développement d'une chaîne d'outils pour la validation de modèles SystemC-TLM en présence d'un ordonnanceur indéterministe, puis son extension pour les modèles discrets avec délais bornés. Cela a nécessité plusieurs contributions complémentaires, qui sont présentées dans ce document, et résumées ci-dessous.

- Identification du problème de l'ordonnancement, via la présentation d'exemples de code incorrect, qui correspondent à des cas fréquents dans les modèles industriels (section 4.3).
- Écriture d'un nouvel algorithme d'arbitrage, robuste aux variations d'ordonnancements pour les modèles avec délais fixes (sous-section 4.4.1 et annexe A).
- Calcul d'une relation de dépendance valide à partir d'une trace d'exécution d'un programme SystemC quelconque (section 5.4). L'analyseur correspondant prend une trace d'exécution en entrée, et renvoie une description de sa classe d'équivalence sous la forme de contraintes d'ordonnancement.
- Description d'un nouvel algorithme, adapté de celui décrit dans [FG05], pour couvrir l'espace des ordonnancements (sous-section 5.4.3). Les ordonnancements générés sont structurés sous la forme d'un arbre de contraintes d'ordonnancements, qui permet d'associer n'importe quel ordonnancement valide à un ordonnancement équivalent et exécuté par notre outil (sous-section 5.5.1).
- Implantation de cet algorithme, et d'un ensemble d'outils auxiliaires dédiés à l'explication des erreurs détectées (chapitre 6).
- Développement, implantation et évaluation sur un exemple réel d'une nouvelle technique pour la validation de modèles transactionnels avec délais bornés (chapitre 8).
- Définition d'un cadre théorique et d'une ébauche pour la réalisation d'un nouveau simulateur SystemC pour machines multiprocesseurs, à la fois efficace, conforme à la spécification et adapté aux modèles TLM fonctionnels (chapitre 9).

Nous avons évalué notre chaîne d'outils, traitant du problème de l'ordonnancement, sur des exemples de nature variée (chapitre 7). Nous avons d'abord détaillé le fonctionnement de notre prototype sur des exemples spécifiquement conçus pour en montrer les limites théoriques, qui se traduisent

par des exécutions redondantes mais rares. Nous l'avons ensuite confronté avec succès à l'exemple fondateur, utilisé pour la présentation et l'évaluation de la réduction d'ordre partiel dynamique. Enfin, nous avons appliqué notre chaîne d'outils à des modèles fournis par STMicroelectronics. Cela a révélé une erreur de synchronisation encore inconnue dans un modèle transactionnel de taille moyenne. Cela a aussi montré deux limitations à notre chaîne d'outils : l'une porte sur la taille maximale des programmes validés, l'autre porte sur l'étape d'instrumentation qui doit être préalablement effectuée. Les outils automatiques conçus pour cette tâche ne sont pas encore au point.

Plusieurs améliorations ont été apportées, notamment la nouvelle preuve, depuis notre première publication [HMMCM06]. Les travaux pour la validation par simulations de modèles avec temps imprécis ont été présentés dans [HMMC06].

Il ne s'agit pas d'outils de vérification formelle permettant de prouver des propriétés sur des modèles transactionnels. Nos générateurs automatiques d'ordonnancements et de jeux de durées ne fournissent de garanties que pour des tests écrits préalablement, et par d'autres moyens. Comme toute approche basée sur la simulation et sans enregistrement d'états, seules les exécutions finies peuvent être traitées.

1.6.2 Plan du document

Les trois premiers chapitres, en dehors de cette introduction, présentent le contexte, l'état de l'art et le problème traité. Les trois chapitres suivants présentent la solution, via la théorie, sa mise en œuvre et son évaluation. Suivent deux chapitres qui traitent chacun d'un problème différent ; l'un est une extension, l'autre est presque indépendant. Enfin, le dernier chapitre fait le bilan des contributions et présente les perspectives envisageables.

L'essentiel des références bibliographiques se trouvent dans le chapitre 2 pour la conception des systèmes sur puce, et dans le chapitre 3 pour leur validation. Les chapitres 8 et 9 possèdent chacun une section pour présenter l'état de l'art qui leur est propre. Tout au long du document, le lecteur trouvera des citations éparses pour accompagner des termes techniques ou rappeler l'origine de certaines découvertes.

Le chapitre 2 décrit le flot de conception des systèmes sur puce. Il insiste sur le rôle des modèles transactionnels, puis poursuit par une présentation de SystemC et de la librairie TLM conçue par l'équipe SPG de STMicroelectronics. Il se conclut par une remarque sur la rupture entre les communications inter-composants et les synchronisations inter-processus, qui est provoquée par la hausse du niveau d'abstraction.

Le chapitre 3 fait le point sur les méthodes existantes pour la validation des modèles SystemC-TLM, les travaux déjà réalisés grâce à la collaboration ST-Verimag. Nous en profitons pour présenter les diverses pistes que nous avons regardées avant de nous concentrer sur le problème de l'ordonnement.

Le chapitre 4 détaille l'ordonnement en SystemC. Des exemples de complexité croissante montrent les problèmes de synchronisation qui nuisent au bon fonctionnement des modèles développés par les ingénieurs. Une nouvelle instruction `yield`, nécessaire pour la suite, `y` est aussi présentée et définie.

Le chapitre 5 constitue le cœur de la thèse. Il explique le principe général, formalise le problème et présente la solution. Il contient notamment un algorithme différent de celui de la littérature, accompagné d'une nouvelle preuve relativement simple.

Le chapitre 6 décrit la chaîne d'outils développée. Cela inclut la mise en œuvre de la théorie, mais aussi la description d'un grand nombre d'outils annexes. Nous verrons que ces derniers sont primordiaux pour la compréhension et la correction des erreurs détectées.

Le chapitre 7 est dédié à l'évaluation de la solution proposée. Il commence par des exemples épurés spécialement conçus pour la mettre en difficulté, continue par un exemple tiré de la littérature et termine par des modèles fournis par STMicroelectronics.

Le chapitre 8 introduit une extension de la technique présentée dans les trois chapitres précédents. Il s'agit là de valider des modèles en présence de temps imprécis. L'extension décrite traite ce nouveau degré de liberté, et fournit les mêmes garanties qu'auparavant. Ce chapitre présente le nouveau problème, définit la solution théorique et son implantation, contient une étude cas et fait un point sur les autres approches similaires déjà existantes.

Le chapitre 9 porte sur un autre sujet de recherche sur lequel nous avons dû nous pencher, à savoir l'accélération des simulations via la réalisation d'un simulateur SystemC multiprocesseur. Nous montrons dans ce chapitre qu'une parallélisation correcte nécessite aussi une analyse des dépendances. Ce chapitre peut être lu indépendamment du reste du document. Il s'agit du début d'un projet récent qui est sous la responsabilité d'un autre membre de l'équipe.

Le chapitre 10 termine cette thèse. Il rappelle d'abord les diverses contributions en matière de recherche théorique, de développement d'outils et de valorisation. Il s'ouvre ensuite sur un ensemble de perspectives prometteuses, incluant principalement des optimisations et des ajouts de fonctionnalités.

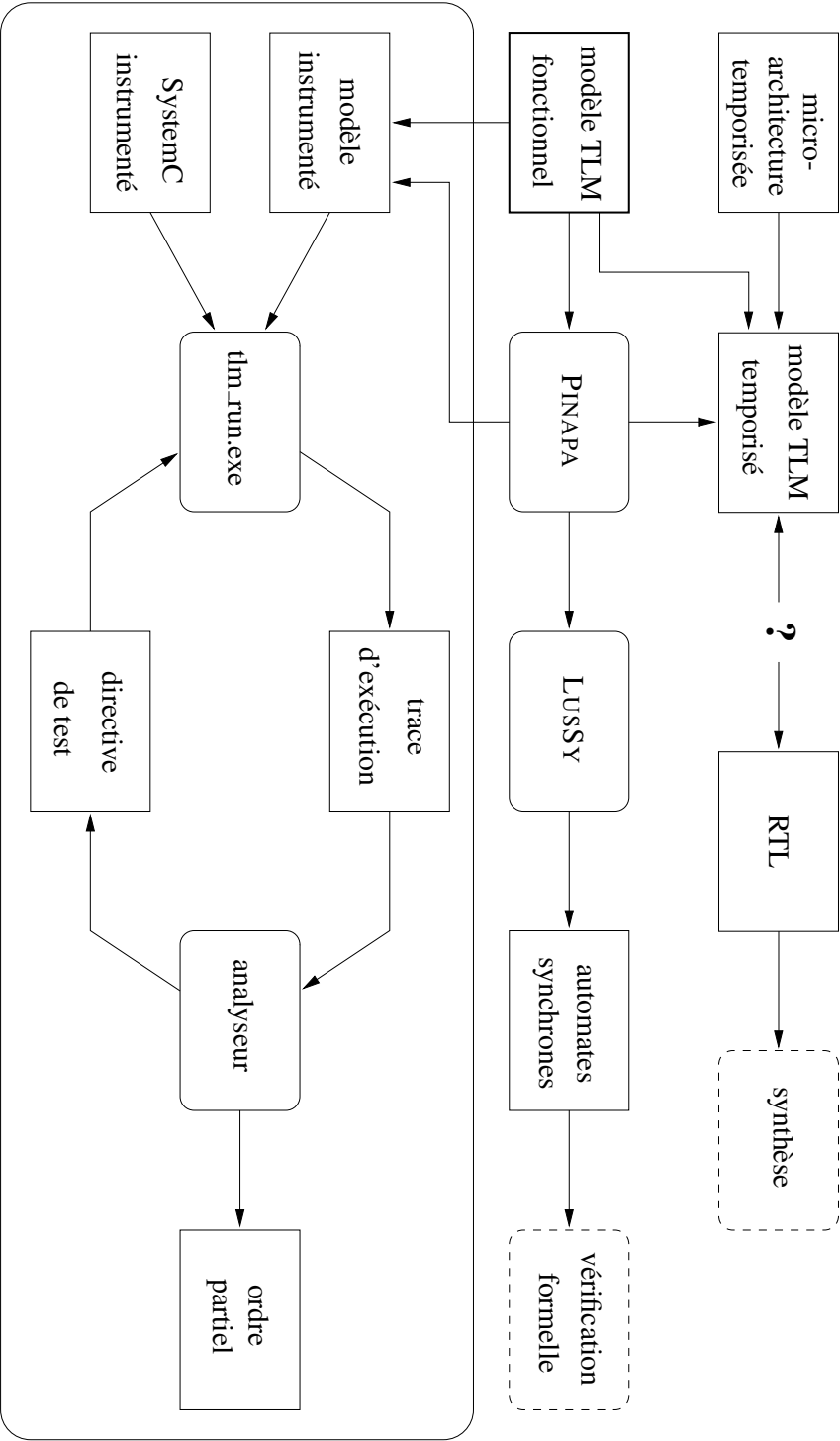


FIG. 1.1 – Schéma général de l’environnement de validation et de développement issu de la collaboration entre l’équipe SPG de STMicroelectronics, et l’équipe Synchrone de Verimag. La partie entourée correspond aux travaux effectués durant cette thèse

Chapitre 2

Modélisation des systèmes sur puce

Sommaire

2.1	Le flot de conception des systèmes sur puce	20
2.1.1	Partitionnement logiciel - matériel	20
2.1.2	Les différents niveaux d'abstraction	21
2.1.3	Validation : vérification, simulation et test	23
2.2	Les modèles transactionnels	25
2.2.1	Concepts communs	25
2.2.2	Les modèles fonctionnels (PV)	27
2.2.3	Les modèles temporisés (PVT)	27
2.3	SystemC et la librairie TLM	28
2.3.1	SystemC	28
2.3.2	La librairie TLM	30

Les systèmes sur puce sont de plus en plus répandus, et se voient confier des tâches de plus en plus complexes. Ils sont de plus soumis à des contraintes non-fonctionnelles fortes portant sur la vitesse, la consommation et le coût. Jusqu'à présent, les capacités physiques des puces progressent suffisamment vite pour faire face aux besoins. Le nombre de transistors par puce augmente ainsi d'environ 50% par an conformément à la loi de Moore, depuis des dizaines d'années ; cette augmentation devrait se poursuivre dans l'avenir, notamment avec la multiplication du nombre de processeurs par puce.

En conséquence, la conception d'un nouveau système sur puce demande un effort de plus en plus important, or la productivité des développeurs avec les méthodes traditionnelles n'augmente que de 30% par an. Un écart se creuse donc entre la capacité physique des puces, et la quantité de code que les développeurs peuvent écrire ; cet écart a été baptisé *design gap*. Pour lutter contre ce problème, le développement de nouvelles techniques de conception est nécessaire. Certaines de ces techniques reposent sur l'utilisation de modèles de haut-niveau, dit *transactionnels*.

Ce chapitre commence par une description du flot complet de conception des systèmes sur puce 2.1, des spécifications informelles à la description finale du matériel. Il se poursuit par une description du niveau de modélisation transactionnel 2.2, et de ses deux principales utilisations : la simulation du logiciel embarqué et l'évaluation de ses performances. Il se termine par la présentation de SystemC et du nouveau standard TLM, qui sont utilisés pour l'implantation des modèles transactionnels 2.3.

2.1 Le flot de conception des systèmes sur puce

La réalisation d'un circuit intégré repose sur sa description *RTL* (de l'anglais : *Register Transfer Level*). Cette description est en effet le point d'entrée des outils de synthèse. Elle décrit comment sont transférées et modifiées les données entre registre à chaque top d'horloge. Un système sur puce n'est pas constitué uniquement de matériel. Il contient aussi du logiciel, et la première étape est de décider ce qui doit être conçu en matériel, et ce qui doit l'être en logiciel.

2.1.1 Partitionnement logiciel - matériel

Il y a eu de véritables progrès dans les techniques de conception de composants matériels. Cependant, le développement du matériel reste plus long et plus coûteux que celui du logiciel. L'idée est donc d'utiliser des composants matériels, suffisamment génériques pour être réutilisables, à la différence des circuits intégrés spécifiques à une application (nommés ASIC, comme *Application Specific Integrated Circuit*). Les composants matériels sont généralement appelés *IP*, comme *Intellectual Property*. Les composants logiciels viennent compléter les fonctionnalités du matériel pour obtenir la fonctionnalité globale voulue.

Les composants logiciels sont aisés à écrire, à corriger, à modifier et à adapter pour une nouvelle utilisation. La correction d'un bug peut se faire à tout moment, et même dans certains cas après que la puce soit entre les mains de l'utilisateur final. Cependant, le logiciel est beaucoup plus lent et consomme beaucoup plus qu'un composant matériel réalisant la même fonctionnalité.

Un compromis doit être trouvé entre une trop forte proportion de logiciel et une trop forte proportion de matériel. Le bon compromis dépend du contexte : un budget restreint va pousser à l'utilisation de composants logiciels alors que des contraintes temporelles sévères peuvent forcer l'utilisation de composants matériels dédiés. Généralement les fonctionnalités élémentaires, et critiques du point de vue de la vitesse, sont gérées par du matériel, alors que les autres fonctionnalités comme l'interface avec l'utilisateur sont conçues avec du logiciel. Le fait de choisir ce qui sera du matériel et ce qui sera du logiciel est appelé le *partitionnement logiciel - matériel*. Le résultat est un système intermédiaire entre un processeur généraliste et un ASIC, contenant plusieurs composants logiciels et matériels communicant et s'exécutant en parallèle. C'est cela que nous appelons *système sur puce*.

L'une des principales tâches du logiciel embarqué est de programmer les composants matériels. Par conséquent, le logiciel est très dépendant du matériel, et ne peut s'exécuter en l'état sur un ordinateur standard. Il est possible de l'exécuter soit sur la puce synthétisée elle-même, soit avec un simulateur utilisant une description du matériel.

Attendre que la première puce soit synthétisée pour exécuter le logiciel embarqué n'est pas une solution. D'une part, afin de réduire le délai avant la mise sur le marché, il est nécessaire de commencer le développement du logiciel embarqué très tôt. D'autre part, l'exécution du logiciel embarqué peut révéler des bugs dans la partie matériel. Si un bug est trouvé en simulant une description du matériel, celui-ci peut être corrigé à moindre coût. En revanche, si le bug est trouvé sur la puce synthétisée, la correction peut coûter très cher. En effet, l'une des premières étapes de la fabrication est la création du masque, et un correctif peut obliger à créer un nouveau masque, ce qui coûte autour d'un million d'euros.

Le plus naturel pour simuler le matériel est d'utiliser la description RTL. Malheureusement, celle-ci est très lente à simuler pour de gros systèmes. La cause vient du haut degré de parallélisme de la description RTL. Le matériel va vite car il se compose d'un très grand nombre de sous-systèmes s'exécutant simultanément. Pour la simulation ceux-ci doivent en revanche être exécutés en séquence, ce qui prend beaucoup de temps. Il est possible d'abstraire certains composants : certaines parties

de la description RTL sont alors remplacées par du code C censé reproduire le même comportement observable. Par exemple, une mémoire peut être simulée par un simple tableau ; un processeur par un simulateur de jeux d'instructions (ISS, *Instruction Set Simulator*). Cette technique, consistant à simuler un mélange de description RTL et de modèle C, est appelée *cosimulation* [Sch03a].

En poursuivant sur la même idée, il est possible de remplacer tous les composants RTL par des modèles en C de plus haut niveau, puis de retirer les horloges. La seule contrainte est que le logiciel embarqué puisse continuer de s'exécuter, sans modification, sur ce nouveau modèle de la puce. Ce nouveau niveau d'abstraction est dit *transactionnel*, ou TLM comme *Transaction Level Model*.

Il existe une autre alternative à la simulation sur modèle abstrait. Il s'agit d'utiliser des systèmes matériels spécifiques appelés *émulateurs matériels*. Tout comme les FPGA, il s'agit de composant matériel *reprogrammable*, c'est-à-dire que les liaisons entre cellules logiques élémentaires peuvent être modifiées électroniquement [Wik06]. Les modèles suffisamment grands pour simuler un système sur puce complexe sont malheureusement très coûteux, et n'offrent que peu de possibilités pour le débogage (contrairement au RTL [HTCT03]). Leur principale utilisation consiste à faire tourner de larges batteries de tests de façon automatique, dans l'espoir de trouver les derniers bugs avant la création du masque (voir par exemple [GNJ⁺96, HZB⁺03]).

La figure 2.1 montre le temps de simulation nécessaire, pour un même calcul, avec les différentes techniques présentées.

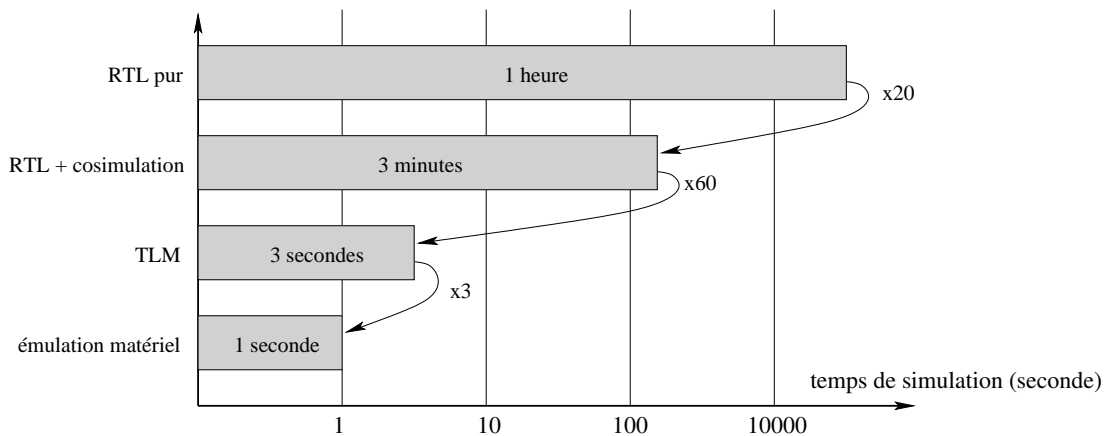


FIG. 2.1 – Temps de simulation nécessaire pour l'encodage et décodage d'une image MPEG4.

2.1.2 Les différents niveaux d'abstraction

La figure 2.2 donne un aperçu du flot de conception. Le point de départ est toujours des spécifications écrites dans un langage informel. Ensuite, une première implantation très abstraite permet de préciser et fixer les fonctionnalités voulues. Puis, une première architecture est proposée et des modèles temporisés permettent de la corriger et de la raffiner. Progressivement, on se rapproche du comportement détaillé de la puce finale. Chaque niveau d'abstraction a ses technologies et ses outils propres ; les principaux noms sont donnés à droite de la figure.

En pratique, le flot n'est pas aussi linéaire, et certains niveaux peuvent être ignorés ou remplacés par des niveaux intermédiaires. La seule description incontournable est celle au niveau RTL, d'où l'on génère automatiquement la description au niveau portes logiques. Pour les niveaux d'abstraction supérieurs, il n'existe pas d'outil automatique pour passer de l'un à l'autre. Les méthodes de raffinement manuel d'une description vers le niveau d'abstraction suivant font actuellement l'objet de

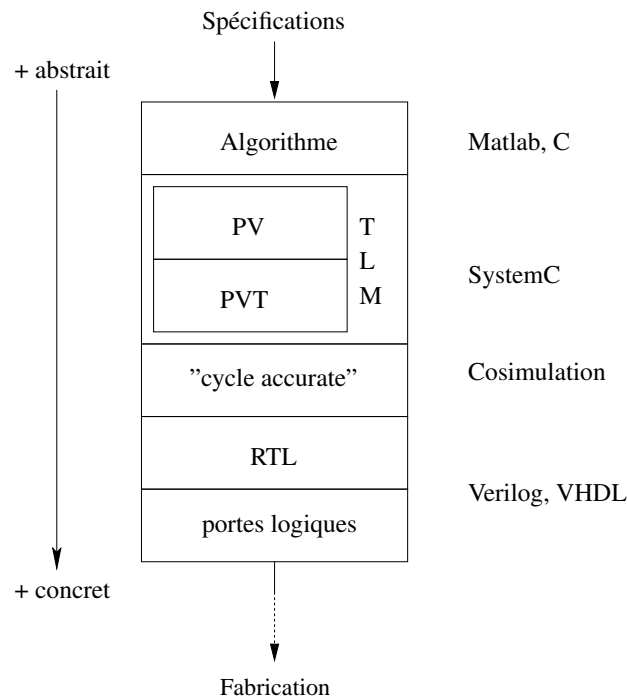


FIG. 2.2 – Les différents niveaux d'abstraction permettant d'aller des spécifications à une description synthétisable.

recherches. En sautant un niveau d'abstraction, on peut donc espérer réduire la quantité totale de code à écrire, et en commençant la description RTL avant que les niveaux supérieurs soient finis, on peut réduire le délai avant la mise sur le marché.

2.1.2.1 Algorithme

Le niveau *algorithme* est le premier à permettre une exécution. La plupart des algorithmes sont souvent déjà disponibles, soit parce qu'ils ont déjà été écrits pour un autre contexte, soit parce qu'ils font l'objet d'une norme et qu'il existe une implantation de référence (comme pour les algorithmes de codage ou décodage vidéo). A ce niveau là, le logiciel et le matériel ne sont pas encore distingués, et le parallélisme n'est pas encore décrit. Ces descriptions sont écrites dans des langages de haut niveau, comme Matlab ou C++. L'une des principales utilités est de préciser et fixer les besoins grâce à des démonstrations au client.

2.1.2.2 Niveau transactionnel : TLM

Au niveau transactionnel, la description de l'architecture et du parallélisme est ajoutée. On distingue les modèles purement fonctionnels, parfois nommés PV (comme *Programmer View*), des modèles temporisés, nommés PVT comme (*Programmer View + Timing*). D'autres niveaux intermédiaires ont été définis [Pas02, CG03].

Les modèles fonctionnels PV permettent la simulation du logiciel embarqué, d'où leur nom. Ils doivent simuler vite afin de ne pas nuire à la productivité des programmeurs du logiciel. A ce niveau d'abstraction, la façon dont sera réalisé chaque module n'est pas encore fixée. Par exemple, tous

les transferts de données peuvent se faire par un même canal de communication, considéré comme physiquement idéal (débit infini).

Pour écrire un modèle transactionnel, il est nécessaire d'avoir déjà une idée du partitionnement logiciel - matériel. Les modèles temporisés TLM-PVT permettent d'évaluer de façon précoce les choix d'architecture et de partitionnement, et donc de revoir certains choix lorsque cela apparaît nécessaire. Afin de préciser les contraintes temporelles sur les sous-systèmes, il est généralement nécessaire de préciser l'architecture et de fixer la taille des mots pouvant circuler sur les canaux de communication. Le débit des bus est alors connu, mais il n'est pas dit comment il sera physiquement obtenu.

2.1.2.3 Niveau "cycle accurate" : CA

L'étape suivante dans le raffinement vers la description RTL consiste à ajouter la notion d'horloge. Un modèle "cycle accurate" décrit ce qui se passe à chaque top d'horloge. Certains détails peuvent encore être décrits sous une forme qui ne permet pas une traduction automatique vers le niveau inférieur, mais globalement toute la micro-architecture est connue. Par exemple, si un processeur ou un bus utilise un pipeline, celui-ci est décrit.

2.1.2.4 Niveau transfert de registre : RTL

Le niveau *transfert de registre* RTL est le plus haut qui permette une synthèse rapide et efficace du circuit intégré. A ce niveau, la valeur de chaque bit à chaque top d'horloge est connu. Les transferts de données peuvent encore se faire par *mot*, un mot étant une donnée de la taille d'un registre. Les langages utilisés ici sont le *VHDL* et *Verilog*.

Il y a grand écart entre les niveaux TLM et RTL. Les modèles TLM ne décrivent pas les solutions matérielles ; ils sont écrits dans des langages impératifs avec une utilisation limitée du parallélisme. Une description RTL est en revanche intrinsèquement parallèle et orienté *flot de donnée*. A moins de pouvoir réutiliser des solutions existantes et génériques, un outil automatique ne peut pas inventer des solutions matérielles efficaces. Certains outils de synthèse commencent à traiter un sous-ensemble du niveau "cycle accurate" [For04], mais il ne faut pas espérer de synthèse efficace de modèles TLM dans un futur proche.

2.1.2.5 Niveau portes logiques, et suivants

A partir d'une description RTL, une description au niveau portes logiques est générée. Celle-ci ne contient plus qu'un réseau de portes logiques *and*, *or*, *not*, Le rôle de ce niveau d'abstraction est comparable à celui de l'assembleur pour le logiciel. Ensuite, des outils de placement et routage se chargent de donner une structure en deux dimensions à ce réseaux. Cette disposition sert alors de base à la construction du masque, d'où l'on tire enfin les premiers circuits intégrés physiques.

2.1.3 Validation : vérification, simulation et test

2.1.3.1 Terminologie

S'assurer de la fiabilité d'un système est l'une des principales préoccupations dans le monde de l'informatique. Ce problème est connu sous le nom de *validation*. Plusieurs méthodologies existent, aussi bien dans le monde du logiciel que dans celui du matériel. Cependant, leurs noms diffèrent.

Dans le monde du logiciel, le fait d'exécuter le programme pour contrôler ses sorties est appelé *test*. En général, une série de tests permet de trouver des erreurs mais ne permet pas de prouver leur

absence. Le problème consistant à *prouver* qu'un programme est correct est appelé *vérification*. Ce terme utilisé seul sous-entend *vérification formelle*. On parle de *vérification semi-formelle* lorsqu'on utilise un mélange de test et de techniques formelles.

Dans le monde du matériel, le terme *vérification* est aussi utilisé mais est quasiment synonyme de *validation* puisque il ne sous-entend pas qu'il s'agit de prouver la correction de la description. Le terme *test* a en revanche un sens complètement différent : il s'agit de vérifier qu'une puce particulière n'a pas de problème physique, autrement dit que la gravure s'est bien passée. Le fait d'exécuter une description pour y trouver des erreurs est appelé *simulation*.

2.1.3.2 Aperçu des méthodologies existantes

Pour chaque puce fabriquée, il faut la *tester*, c'est-à-dire s'assurer de l'absence de défauts physiques liés à sa fabrication. C'est équivalent à vérifier que les feuilles d'un livre sont bien découpées, et qu'il ne manque pas d'encre ; mais il ne s'agit pas de vérifier le contenu, comme par exemple l'orthographe. Pour tester une puce, il faut stimuler ses parties internes et observer les sorties. Cela oblige à intégrer à la puce des systèmes matériels dont le seul rôle est de permettre son test. Cela est appelé BIST, comme *Built-In Self Test*. Cette portion de la puce ne sera plus utilisée passé la phase de test. Elle n'est pas décrite dans les modèles abstraits car elle n'y a aucune utilité.

Le logiciel embarqué est développé sur les modèles transactionnels. Normalement, il est donc prêt lorsque les premières puces sortent de la fabrication. Cependant, les vérifications finales doivent avoir lieu sur le circuit intégré final. Cela permet soit de corriger un bug du logiciel, soit dans certains cas de modifier le logiciel pour contourner un bug du matériel.

Le principal objectif est d'éviter de devoir modifier la description RTL alors qu'un masque, très coûteux, a déjà été fabriqué. Les logiciels de synthèses à partir du RTL (coûteux, eux aussi) peuvent raisonnablement être considérés comme corrects.

La vérification formelle est utilisée autant que possible. La technique la plus utilisée pour cela est la *vérification par modèle* ou *model checking*. La nature essentiellement booléenne de ces descriptions se prête bien à l'utilisation des méthodes symboliques comme les BDD ou les solveurs SAT, qui permettent de couvrir exhaustivement l'espace d'état sans les énumérer tous (outils : [HLR92, YS01, CCG⁺02], études de cas : [Sch03b, YG02]). Cependant, la taille des descriptions des systèmes sur puce fait que la vérification se heurte souvent au problème de l'*explosion du nombre d'états*. Certains composants, et l'assemblage des composants, doivent donc être validés par des techniques de simulations.

Un jeu de tests pour un système sur puce peut être très grand. Il peut comporter des tests générés automatiquement mais la plupart sont écrits à la main par des équipes d'ingénieurs spécialisées. Simuler tous ces tests peut demander plusieurs jours, même en distribuant le calcul sur plusieurs machines. Il n'est pas réaliste de vérifier tous les résultats à la main. Des *oracles* automatiques doivent donc être définis. Ceux-ci peuvent vérifier que les sorties respectent un ensemble de *propriétés*, généralement décrites dans une *logique temporelle* [BBP89, LMTY02]. Une autre solution est de comparer les résultats à une *implantation de référence*.

2.1.3.3 Modèle de référence

Il existe plusieurs solutions pour comparer une simulation d'une description RTL à une exécution d'un modèle plus abstrait, dit de *référence*. L'une d'elles consiste à observer les communications. Cela suppose que les canaux de communications soient semblables entre les deux niveaux d'abstraction. Une autre solution, très répandue, consiste à comparer l'état de la mémoire de la description RTL avec

celle du modèle de référence, à la fin de l'exécution ou en d'autres points bien définis. Cette deuxième solution évite certains problèmes techniques, comme de devoir faire tourner ensemble du code VHDL avec du code SystemC, par exemple. Une fois les tests construits et simulés, la difficulté consiste à bien définir ce qui est pertinent dans les observations effectuées, de ce qui ne l'est pas.

Le modèle servant d'implantation de référence doit être suffisamment abstrait pour être aisé à écrire et à valider, mais aussi suffisamment concret pour que ses résultats soient comparables avec ceux du RTL. Les modèles fonctionnels TLM-PV sont souvent de bons candidats, ou les modèles temporisés TLM-PVT si l'on souhaite avoir une granularité plus proche de celle du RTL.

2.2 Les modèles transactionnels

Les modèles transactionnels décrivent l'architecture et le comportement du matériel sous forme de processus concurrents. Les trois principales utilisations sont les suivantes :

- simulateur pour le développement du logiciel embarqué ;
- prototype pour l'évaluation des performances non-fonctionnelles ;
- implantation de référence pour la validation du RTL.

Selon leurs utilisations, ils peuvent être purement fonctionnels ou temporisés.

2.2.1 Concepts communs

Nous résumons d'abord les principaux concepts des modèles TLM, tels qu'ils sont en train d'être normalisés par l'OSCI¹ [t1m06]. Une description complète de l'approche TLM peut être trouvée dans le livre [Ghe05] : *Transaction-Level Modeling with SystemC. TLM Concepts and Application for Embedded Systems*.

Les modèles TLM représentent tout d'abord une architecture ; La figure 2.3 en donne un exemple.

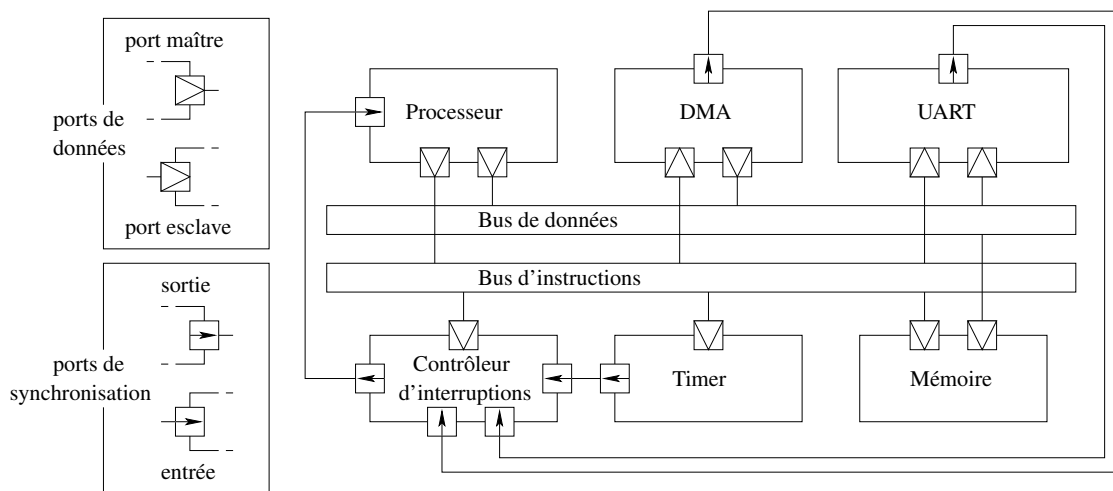


FIG. 2.3 – Exemple d'architecture d'un modèle TLM.

L'architecture est définie par un ensemble de composants, reliés entre eux par des canaux de communication. On distingue deux types de canaux de communication : les canaux transactionnels pour l'échange de données et d'informations, et les canaux de synchronisation permettant juste de

¹Open SystemC Initiative

notifier un événement à un autre composant. Chaque composant peut contenir zéro, un ou plusieurs processus.

Une transaction constitue un échange *atomique* de données entre un composant *initiateur* A et un composant *cible* B. L'initiateur est celui qui prend l'initiative de la transaction. Par analogie avec les applications internet, l'initiateur correspond au client et la cible au serveur. L'initiateur est parfois aussi appelé *maître*, et la cible *esclave*. Un *port initiateur*, qui permet d'initier une transaction, est toujours relié à un *port cible*, qui permet de la recevoir. Certains composants possèdent uniquement des ports initiateurs, comme les processeurs ; d'autres, comme les mémoires, possèdent uniquement des ports cibles. Enfin, certains composants possèdent les deux types de port, à l'exemple du DMA² que l'on programme via son port cible et qui accède à la mémoire via son port initiateur.

Les données ne circulent pas forcément de l'initiateur à la cible. Cela dépend de la *nature de la transaction* ; en général, il s'agit soit d'une *lecture*, soit d'une *écriture*. Cette *nature* est l'une des informations qui composent une transaction. La liste des informations composant une transaction est définie par un protocole. Il existe plusieurs protocoles mais la plupart définissent les informations suivantes :

- La *nature*, valant généralement *read* ou *write* ;
- L'*adresse*, généralement codée par un entier, qui détermine d'une part le composant cible, et d'autre part quel mot ou registre de ce composant est visé ;
- La *donnée* que l'on veut transmettre ou recevoir ;
- Des *méta-données* pouvant contenir : un statut de retour, des infos de durée et toutes autres informations pouvant servir au contrôle ou au debug.

Les bus sont des composants spécifiques qui se chargent de transférer les transactions qu'ils reçoivent vers les composants cibles. Pour cela, ils se basent sur l'adresse contenue dans la transaction, et sur la *carte des adresses mémoires* (ou : *memory map*), qui associe à chaque port cible une plage d'adresse (cf figure 2.4). Il est possible de coder un bus comme un autre composant, mais en pratique les modèles de bus sont fournis avec le protocole.

; Module name	start address	size
DMA.target_port	0x04c00	0x00080
Memory.data_port	0x80000	0x20000
...		

FIG. 2.4 – Extrait d'une carte des adresses mémoires

La taille de la donnée est variable et dépend du protocole. Selon le niveau d'abstraction utilisé, le transfert d'une image peut par exemple se faire pixel par pixel, ligne par ligne ou en une seule fois. Inversement, certains protocoles permettent d'échanger des informations dont la taille est inférieure au mot ; cela ce fait grâce au mécanisme de *byte enable* (littéralement : *octet actif*).

Le comportement du modèle est défini par des processus écrits dans un langage de haut niveau comme C++. Ils peuvent effectuer des calculs et communiquer en initiant des transactions, en réagissant à des requêtes ou en notifiant des interruptions. Un modèle est dit *transactionnel* si une transaction peut être initiée par un processus en une seule instruction. Cela les distinguent des modèles de plus bas niveau dans lesquels une transaction nécessite plusieurs étapes, par exemple : écriture de l'adresse sur les signaux, puis écriture de la donnée au top d'horloge suivant, attente de la confirmation sur un autre signal, ...etc.

²abréviation de *Direct Memory Access*

2.2.2 Les modèles fonctionnels (PV)

Les modèles fonctionnels, baptisés PV comme “*Programmer View*”, se destinent généralement à la simulation du logiciel embarqué. Ils doivent être suffisamment concrets pour que le logiciel puisse fonctionner comme sur la puce finale, sans retouche du code ; et ils doivent être suffisamment abstraits pour que la simulation soit rapide.

Simuler le logiciel suppose que les composants matériels soient connus, ainsi que leur interface. L’interface d’un composant est définie par une liste de ports de différentes natures. Les ports transactionnels cibles permettent généralement d’accéder à des registres, qui permettent de programmer le composant pour lui faire exécuter une requête, comme par exemple le décodage d’une image.

L’intérieur des composants est en revanche très différent de la puce finale. Les traitements sont réalisés par des algorithmes optimisés pour le logiciel. Tout ce qui concerne les propriétés non-fonctionnelles est ignoré ; les composants sont en quelque sorte physiquement idéaux : les temps de réponse, de traitement et les débits peuvent être considérés comme nuls. Cela évite les interactions implicites entre processus, comme par exemple les embouteillages pour les accès à la mémoire.

Tous les échanges d’informations ou de données entre composants ou processus doivent en revanche être explicitement synchronisés. Comme aucune durée n’est connue, il ne suffit pas d’attendre un certain temps pour être sûr qu’une donnée est disponible. Les synchronisations se font grâce à des variables partagées et des événements, éventuellement véhiculés entre les composants par des fils d’interruption.

La programmation de systèmes asynchrones est reconnue comme étant plus difficile que celle des systèmes synchrones, surtout s’ils doivent être indépendants aux délais [RR02]. La cause principale vient de leur nature intrinsèquement indéterministe. Il n’est donc pas surprenant que les modèles partiellement temporisés remplacent souvent les modèles purement fonctionnels. Cela permet en effet de simplifier la conception de ces modèles, tout en respectant le cahier des charges pour la simulation du logiciel embarqué.

2.2.3 Les modèles temporisés (PVT)

Le niveau d’abstraction suivant dans le flot de conception consiste à ajouter des informations approchées sur les capacités physiques des composants. Ajouter des annotations de durée sur les instructions ou groupes d’instructions n’est pas suffisant. En effet, les propriétés temporelles dépendent aussi beaucoup de l’architecture. Les modèles temporisés ont été baptisés PVT, comme “*Programmer View plus Timing*”

Au niveau fonctionnel, un modèle de bus générique est par exemple suffisant pour tous les modèles. en revanche, pour évaluer le débit et les temps de transfert, des modifications spécifiques sont nécessaires. Tout d’abord, il faut fixer la *largeur* du bus, par exemple : 32 bits. Cela oblige à découper en plusieurs les transactions dont la donnée est plus large ; autrement dit, il faut réduire la *granularité* des communications. Ensuite, il faut tenir compte de l’architecture du bus. Il peut y avoir deux bus distincts, l’un pour les données, l’autre pour les instructions. Le bus peut aussi utiliser un *pipeline*, ou même constituer un véritable *réseau*. Pour chaque composant, la modélisation de ces informations est indispensable pour une évaluation précise des performances temporelles du modèle complet.

Un sujet de recherche actif concerne la conception d’un modèle PVT à partir du modèle PV du même composant. Modifier le code PV pour y ajouter les informations nécessaires n’est pas une solution satisfaisante car cela permettrait de changer la fonctionnalité et tout le code devrait alors être validé à nouveau. L’idée est de décrire les informations temporelles ainsi que la nouvelle granula-

rité des communications dans un modèle T distinct, que l'on peut voir comme un *aspect* d'un type bien particulier [CMMC07]. Les modèles PV et T sont reliés statiquement par un squelette généré automatiquement, et leurs exécutions sont ensuite *dynamiquement tissées* pour former une exécution PVT. La formalisation de l'approche devrait mener à une *preuve par construction* démontrant que la fonctionnalité est bien conservée, moyennant le respect de certaines règles.

La simulation des modèles temporisés est généralement basée sur une notion de temps globale. Toutefois, il existe aussi des techniques de modélisation et simulation basées sur des horloges locales à chaque composant. Dans [VPG06], le canal de communication se charge de resynchroniser les différentes horloges.

Il serait intéressant de pouvoir évaluer d'autres propriétés non-fonctionnelles, comme la consommation énergétique, à partir des modèles TLM. Cependant, aucun projet n'a, à ce jour, donné de résultats convaincants. Il semble que les informations nécessaires soient encore absentes à ce niveau d'abstraction ; des techniques d'analyses précises au niveau "cycle accurate" existent d'ores et déjà [BNG⁺06].

2.3 SystemC et la librairie TLM

La librairie TLM, permettant d'écrire des modèles du même type, est basée sur SystemC, que nous allons donc présenter en premier.

2.3.1 SystemC

2.3.1.1 Présentation des principales caractéristiques

Un langage pour la modélisation de système matériel doit respecter au minimum les trois caractéristiques ci-dessous :

- une grande vitesse de simulation ;
- une structuration en composants afin de faciliter la réutilisation ;
- une sémantique d'exécution parallèle.

De plus, une librairie implantant les structures fréquentes dans le matériel doit être disponible.

Le langage C++ répond aux deux premiers critères : la vitesse de simulation est assurée par une compilation efficace et la réutilisabilité est fournie par la couche objet. Cependant, il n'offre pas de solution simple pour la programmation parallèle. L'approche suivie par SystemC a été de compléter le langage C++ avec un mécanisme pour l'exécution parallèle de processus, ainsi qu'une librairie très fournie pour la modélisation du matériel.

SystemC est implanté sous la forme d'une librairie. Un programme SystemC est donc un programme C++. Cela a plusieurs avantages. Premièrement, cela permet un apprentissage rapide pour les nouveaux programmeurs. Deuxièmement, cela permet d'utiliser les outils d'édition déjà existant pour C++.

De plus, l'utilisation du langage SystemC est libre, et une implantation *open-source* est fournie. Cela évite d'être dépendant d'un fournisseur d'outils de conception, favorise la diffusion des modèles décrits dans ce langage, et, ce qui nous intéresse plus particulièrement, facilite les modifications à des fins d'expérimentations. Il existe aussi des implantations propriétaires, comme NC-SystemC [Cad99] qui améliore les possibilités de cosimulation avec du VHDL.

La version 2.1 de SystemC est désormais définie par un standard IEEE [Ope05], qui a été écrit en concertation avec les principaux acteurs du domaine. Ce langage est désormais très répandu, et de

nombreux outils ont été développés autour ou par-dessus, notamment pour la visualisation graphique des communications entre composants.

2.3.1.2 Alternatives à SystemC

Le langage SpecC [DGG06], lui aussi open-source, a suivi une autre approche : celle de créer un nouveau langage, inspiré de C et C++, avec les fonctionnalités requises pour la modélisation de systèmes matériels. Cette approche n'a pas eu beaucoup de succès dans l'industrie, principalement parce qu'elle ne permet pas de réutiliser des outils génériques, par exemple pour le débogage.

D'autres approches plus formelles ont été proposées pour la description de systèmes hétérogènes matériel et logiciel, comme par exemple Metropolis [BWH⁺03]. L'idée est là d'avoir une description formelle du langage utilisé. Cela facilite la vérification formelle, d'une part en retirant toute ambiguïté de la sémantique, d'autre part en limitant mieux ce qu'il est possible d'écrire.

Il existe aussi des langages spécifiques à la description d'architecture, et qui ignorent le comportement des composants [SPI03]. L'objectif est alors d'uniformiser la description des interfaces entre composants afin d'améliorer la réutilisabilité.

2.3.1.3 Structure d'un modèle SystemC

SystemC permet de modéliser à la fois l'architecture et le comportement du matériel, ainsi que le logiciel embarqué. Il est conçu pour être compatible avec tous les niveaux d'abstraction du flot de conception d'un système sur puce.

Les composants sont décrits grâce à une classe `sc_module` de laquelle on hérite. Cette classe peut contenir des déclarations de processus, et des ports que l'on relie ensuite aux canaux de communication. Les figures 2.6 et 2.7 donnent le code d'un programme SystemC, décrivant deux modules dont l'un est instancié deux fois. L'architecture correspondante est schématisée par la figure 2.8. Les connexions entre ports et canaux sont réalisées après l'instantiation des modules (lignes 66 à 70). Il est à noter que les modules SystemC sont hiérarchiques : ils peuvent aussi contenir d'autres modules et connexions formant un sous-système.

Les processus sont décrits par des méthodes de classes `sc_module`, sans argument et sans type de retour (lignes 8-12 et 24-37). En plus des instructions C++ normales, ils disposent d'instructions `wait` pour se mettre en attente (ligne 34), et d'événements `sc_event` pour réveiller d'autres processus.

Deux types de connexion entre modules sont à distinguer :

1. les connexions via un canal de communication ;
2. les connexions directes entre deux modules.

Dans le premier cas, les processus de chacun des deux modules appellent des fonctions du canal de communication via des ports de la classe `sc_port` (cf figure 2.5). Un canal de communication répond à des appels de fonctions et peut notifier des événements aux processus pour les synchroniser ; il stocke des données. Les plus utilisés sont les signaux `sc_signal` et les files `sc_fifo`, mais il en existe de nombreuses variantes et l'utilisateur peut créer ses propres canaux en héritant de la classe `sc_prim_channel`.

Dans le second cas, le processus initiateur appelle directement une fonction d'un autre module via un port de la classe `sc_export`³ (cf figure 2.9). Les composants bus sont aussi modélisés par des modules SystemC. Par conséquent, les liaisons de composants à bus et de bus à composants sont aussi

³Historiquement cette classe devait servir pour les modules hiérarchiques, en permettant l'accès à des sous-modules ; mais son usage a ensuite été étendu pour les connexions entre modules disjoints.

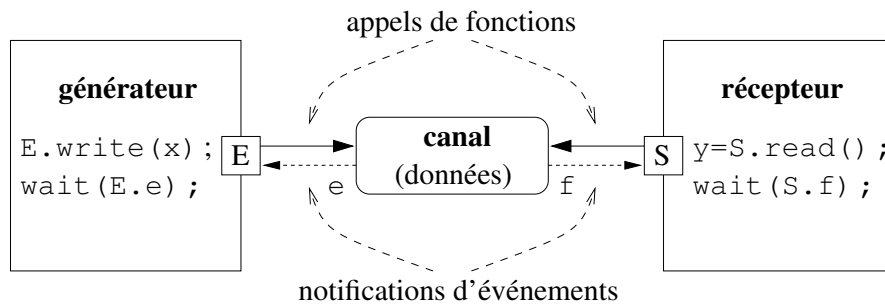


FIG. 2.5 – Connexions via un canal de communication.

classées dans cette catégorie. Grâce à ce mécanisme un processus peut exécuter du code d'un autre module. Cela réduit le nombre d'opérations nécessaires pour une communication particulière, mais oblige à synchroniser les processus d'une autre manière.

Lors de l'exécution du modèle, la première phase, dite d'*élaboration*, consiste à instancier les modules ainsi que leurs processus, et à connecter les différents ports. A la fin de cette phase, l'architecture du modèle est construite et la *simulation* proprement dite peut alors commencer. Cela est fait par un appel à fonction `sc_start` qui rend la main au noyau SystemC (ligne 71 de l'exemple, l'argument permet de borner la longueur de la simulation). Les synchronisations entre processus et leur ordonnancement est détaillé plus loin, à la section 4.1.

2.3.2 La librairie TLM

Les communications dans les modèles TLM développés à STMicroelectronics n'utilisent que des liaisons directes de module à module (donc éventuellement via un module bus, mais sans canal dérivant de `sc_prim_channel`). Les premiers modèles TLM utilisaient les liaisons par canal `sc_signal` pour véhiculer les interruptions, mais depuis peu ces liaisons ont aussi été remplacées par des liaisons directes.

2.3.2.1 La fonction `transport` et les protocoles

Le cœur de la librairie TLM est désormais constitué par la déclaration ci-dessous :

```
template<REQ, RSP>
class tlm_transport_if: public sc_interface {
public:
    virtual RSP transport(const REQ&) = 0;
};
```

Comme son nom l'indique cette fonction sert à transporter une transaction, d'un port initiateur à un module cible. Bien entendu, cette simple fonction n'est pas suffisante pour écrire un modèle TLM. Pour cela, il faut disposer d'un *protocole* qui instancie cette déclaration en précisant le contenu REQ de la transaction et le type de réponse RSP.

Le protocole fournit aussi des fonctions construites par dessus la fonction `transport`. Ce sont elles qui seront utilisées par le développeur du modèle. Les plus fréquentes sont `read` et `write`. Ces fonctions sont des méthodes du port initiateur; elles se chargent de construire un objet de type REQ avec la nature correspondante (READ ou WRITE), et les informations provenant de ses arguments, puis d'appeler la fonction de transport dessus. La fonction `transport` est implantée pour appeler une

```
1  #include "systemc.h"
2  #include <iostream>
3  #include <vector>
4
5  struct module1 : public sc_module {
6      sc_out<bool> port;
7      bool m_val;
8      void code1 () {
9          if (m_val) {
10             port.write(true);
11         }
12     }
13     SC_HAS_PROCESS(module1);
14     module1(sc_module_name name, bool val)
15         : sc_module(name), m_val(val) {
16         // enregistre "void code1()"
17         // comme étant un SC_THREAD (processus standard)
18         SC_THREAD(code1);
19     }
20 };
21
22 struct module2 : public sc_module {
23     sc_in<bool> ports[2];
24     void code2 () {
25         std::cout << "module2.code2"
26             << std::endl;
27         int x = ports[1].read();
28         for(int i = 0; i < 2; i++) {
29             sc_in<bool> & port = ports[i];
30             if (port.read()) {
31                 std::cout << "module2.code2: exit"
32                     << std::endl;
33             }
34             wait(); // attente sans argument, utilise
35                 // la liste de sensibilité statique.
36         }
37     }
38     SC_HAS_PROCESS(module2);
39     module2(sc_module_name name)
40         : sc_module(name) {
41         // enregistre "void code2()"
42         // comme étant une SC_METHOD (processus simplifié)
43         SC_METHOD(code2);
44         dont_initialize();
45         // liste de sensibilité statique pour code2
46         sensitive << ports[0];
47         sensitive << ports[1];
48     }
49 };
```

FIG. 2.6 – Exemple de programme SystemC : définition des modules.

```

50 // la vraie fonction main est dans la librairie SystemC
51 int sc_main(int argc, char ** argv) {
52     bool init1 = true;
53     bool init2 = true;
54     if (argc > 2) {
55         init1 = !strcmp(argv[1], "true");
56         init2 = !strcmp(argv[2], "true");
57     }
58     sc_signal<bool> signal1, signal2;
59     // instantiation des modules
60     module1 * instance1_1 =
61         new module1("instance1_1", init1);
62     module1 * instance1_2 =
63         new module1("instance1_2", init2);
64     module2 * instance2 =
65         new module2("instance2");
66     // connexion des ports de modules et des canaux
67     instance1_1->port.bind(signal1);
68     instance1_2->port.bind(signal2);
69     instance2->ports[0].bind(signal1);
70     instance2->ports[1].bind(signal2);
71     sc_start(-1);
72 }

```

FIG. 2.7 – Exemple de programme SystemC : fonction principale.

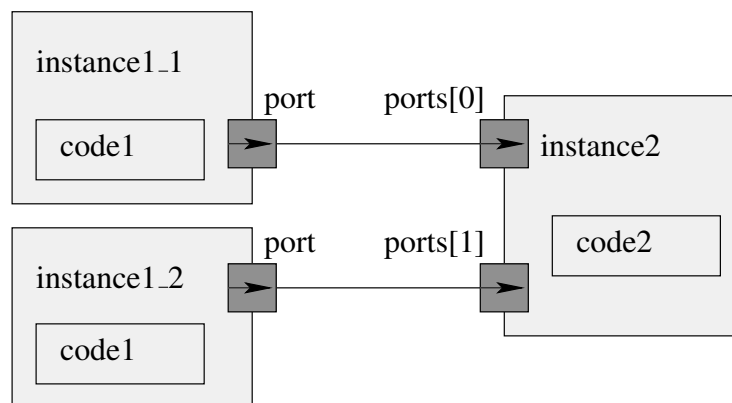


FIG. 2.8 – Aperçu graphique de l'architecture de l'exemple SystemC.

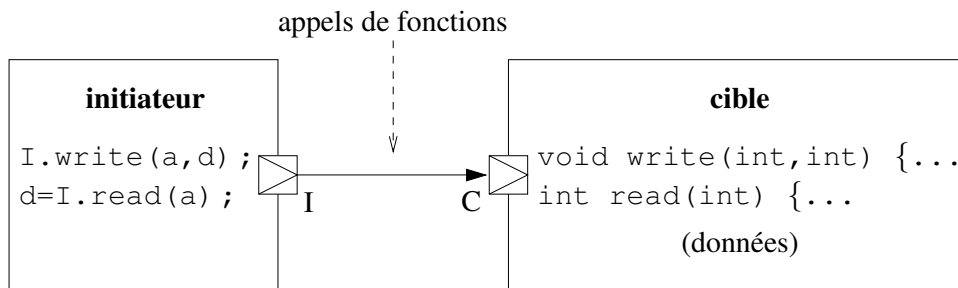


FIG. 2.9 – Connexions directes entre modules.

méthode du module cible qui porte le même nom. Pour le développeur, tout se passe comme s'il avait directement appelé la méthode du module cible.

Enfin, les protocoles sont souvent distribués avec des modèles de bus. Le comportement de ces modèles de bus dépend du niveau d'abstraction. Pour les protocoles dédiés aux modèles fonctionnels, le modèle de bus est souvent appelé *router* car il se contente de transmettre la transaction en fonction de son attribut adresse, sans modéliser les interactions liées à l'utilisation du bus. Dans les niveaux d'abstractions inférieurs (par exemple TLM-PVT), les modèles de bus ordonnent les différentes transactions pour pouvoir simuler son encombrement. Ils peuvent aussi proposer des fonctions spécifiques, comme par exemple `lock` et `unlock` qui permettent à un processus de se réserver le bus afin de réaliser une séquence atomique de transactions.

Dans cette thèse, nous nous intéressons essentiellement aux modèles fonctionnels. Ceux-ci utilisent deux protocoles : *TAC* et *tlm_synchro*. Le protocole TAC sert aux transactions proprement dites, c'est-à-dire aux échanges de données entre composants via un bus. Il fournit un modèle de bus : le `TAC.Router`. Le protocole *tlm_synchro* est destiné à la modélisation des fils d'interruptions. Ce protocole est générique (*template*) sur le type de donnée transportée, mais il est principalement utilisé avec le type booléen. Comme cela modélise des liaisons directes entre composants matériels, il n'y a ni adresse, ni bus.

2.3.2.2 Comparaison des interfaces composants et des interfaces processus

Un canal de communication, par exemple un signal SystemC, est à la fois une interface entre deux composants (car il relie deux composants ou plus), et entre processus (car les processus connectés à ses ports peuvent s'échanger des informations en appelant les fonctions du canal). Si un modèle est structuré avec un seul processus par module, et uniquement des connexions via des canaux de communication, alors les interfaces composants correspondent exactement aux interfaces processus. Cela simplifie beaucoup de choses, à commencer par la compréhension globale du modèle. Ce type de structure est utilisé pour les descriptions bas niveau (RTL ou *cycle accurate*).

Les modèles transactionnels n'utilisent pas ces canaux de communication pour connecter les modules. Lors d'une transaction, un processus exécute du code de son module d'origine, puis du code du module bus et enfin du code du module cible ; s'il le souhaite, il peut même recommencer une nouvelle transaction sans laisser les autres processus s'exécuter. Une connexion directe entre deux modules (dont l'un peut être un bus) matérialise une interface entre composants, mais n'est pas une interface entre processus. Les interfaces entre processus se situent ailleurs : à l'intérieur des modules.

Considérons par exemple un système composé de deux composants maîtres et d'une mémoire, reliés par un bus `TAC.Router`. Les composants maîtres contiennent chacun un processus. La mémoire est modélisée par un grand tableau, situé dans un module esclave sans processus. Les deux

processus écrivent dans la mémoire en effectuant des transactions via le bus. Il y a alors :

- trois interfaces entre les modules, rassemblées autour du bus, et matérialisées par les ports initiateurs et cibles ;
- une interface entre les deux processus, dans le module mémoire, et matérialisée par un tableau partagé.

Il est aussi possible de considérer que le module modélisant le bus constitue une seule interface entre plusieurs composants ; c'est juste une question de terminologie. Par contre, il ne faut pas oublier les interfaces composants constitués par les fils d'interruptions. Avec le protocole `tlm_synchro` actuel, ces interfaces ne sont pas non plus des interfaces processus (contrairement aux signaux qu'ils ont remplacés). Les événements SystemC `sc_event` ne sont utilisés qu'à l'intérieur des modules, et servent à la communication entre processus. La figure 2.10 représente l'ensemble des interfaces pour un petit système, inspiré d'un exemple de STMicroelectronics et étendant celui décrit ci-dessus.

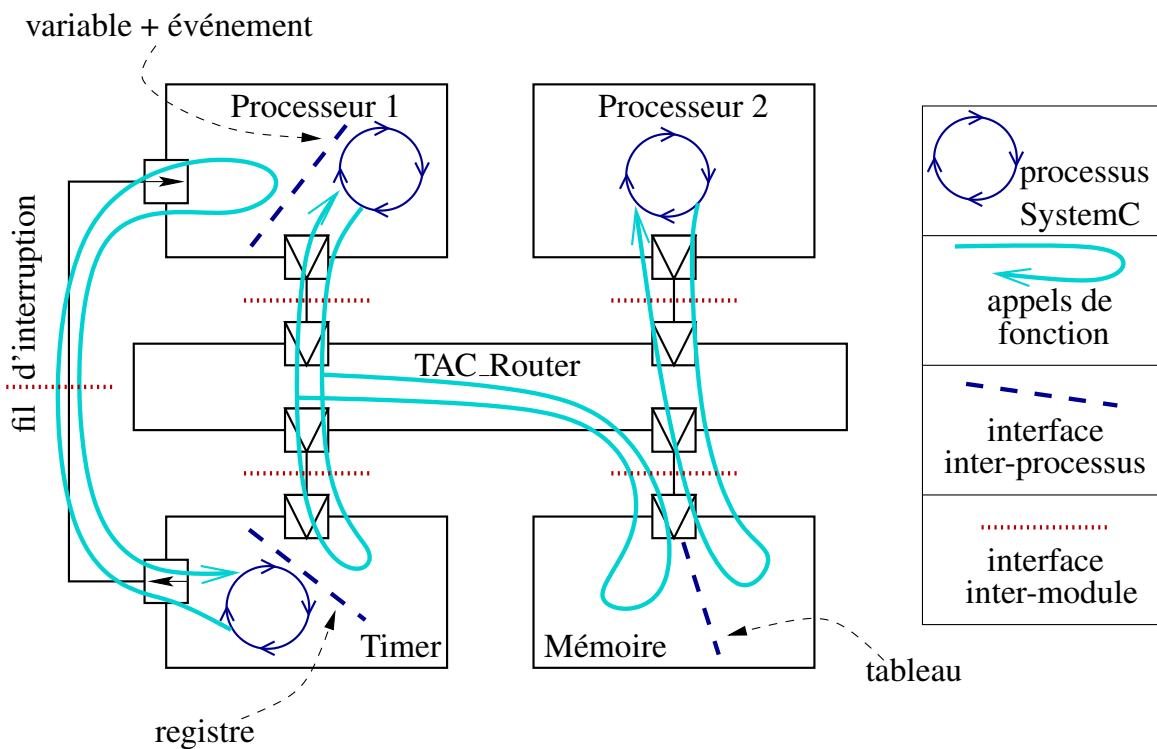


FIG. 2.10 – Exemple de modèle TLM-PV et ses interfaces.

Cette thèse porte sur l'étude des synchronisations entre processus. Nous allons donc parler essentiellement des interfaces processus, et très peu des transactions. Cependant, les problèmes étudiés sont indirectement mais fortement liés au niveau d'abstraction TLM.

Chapitre 3

Validation des modèles transactionnels

Sommaire

3.1 Vérification formelle	35
3.1.1 Outils candidats pour la validation des modèles de SoCs	36
3.1.2 La chaîne d’outil LUSY	36
3.2 Validation par simulations	37
3.2.1 Environnement de tests	37
3.2.2 Outils pour la génération de tests	39
3.3 Combinaison des méthodes	40
3.3.1 Génération de tests ciblant un trou de couverture	41
3.3.2 Vérification à la volée et simulations équivalentes	42

L’objectif essentiel de la validation est, soit de garantir qu’un programme est correct, soit de donner un contre-exemple. Pour cela, nous cherchons à construire des outils aussi automatiques que possible. Bien sur, les propriétés qui doivent être vérifiées par le programme, ainsi que les contraintes devant être respectées par les entrées, ne peuvent être inventées par la machine. Un outil de vérification prend donc au minimum deux entrées : un programme et une spécification. Le format de la spécification est très variable ; elle est parfois intégrée au programme sous forme d’annotations et assertions.

Cette thèse s’inscrit dans un ensemble plus large de travaux dont le but est la validation des modèles transactionnels écrits en SystemC-TLM. Dans ce court chapitre, nous présentons les méthodes existantes ou envisageables : la vérification formelle (section 3.1), la validation par simulation (section 3.2) et enfin les méthodes hybrides (section 3.3). C’est dans cette dernière catégorie que se situe les recherches effectuées durant les trois années de cette thèse.

3.1 Vérification formelle

La vérification formelle regroupe les techniques *automatiques* permettant de *prouver* qu’un programme respecte sa spécification. Généralement, elle permet aussi de trouver un contre-exemple, ou de donner des indications pour en trouver un. Il s’agit donc de la meilleure solution, lorsque celle-ci est applicable. Prouver la correction d’un programme est *indécidable* dans le cas général, mais des *abstractions conservatrices*, c’est-à-dire qui préservent les erreurs mais peuvent en ajouter de fausses, permettent souvent de se ramener à un cas décidable, comme par exemple celui des automates avec

nombre d'états fini. Malheureusement, même dans le cas décidable, le coût reste exponentiel en fonction de la taille du programme, et ce malgré toutes les optimisations apportées.

Une autre alternative pour prouver qu'un programme est correct est d'utiliser un *assistant de preuve*, qui vérifie une preuve écrite manuellement (voir par exemple [Phi03] pour le matériel). Le principal inconvénient de ces méthodes est le coût de développement des logiciels ainsi prouvés. Écrire les preuves de façon rigoureuse prend en effet beaucoup de temps. Les modèles transactionnels ne sont pas considérés comme suffisamment critiques pour justifier ce surcoût.

3.1.1 Outils candidats pour la validation des modèles de SoCs

Les modèles SystemC décrivent avant tout des systèmes matériels. Une première idée est donc d'utiliser les outils déjà existants pour la vérification du matériel. Cependant, ceux-ci sont conçus pour des descriptions aux niveaux RTL, et reposent sur des caractéristiques de ce niveau d'abstraction (horloge globale, absence de pointeurs, prédominance des booléens ...), que l'on ne retrouve plus dans les modèles transactionnels. En dehors de LusSy, dont nous discuterons plus loin, les outils conçus pour la vérification de programmes SystemC ne traitent qu'un sous-ensemble du langage correspondant à celui qui permet d'écrire du RTL (cf [DG03]).

Un modèle SystemC est aussi un programme C++. Une autre idée est donc de regarder du côté des outils de vérification pour le logiciel, et plus particulièrement pour le C ou C++. Des outils comme CBMC[CKL04] ou SLAM[BR00] permettent de traiter des programmes avec des structures de données dynamiques et des algorithmes complexes. Ils ne sont, en revanche, pas prévus pour traiter le parallélisme, or celui-ci est très présent dans les modèles SystemC. De plus, le parallélisme de SystemC est assez spécifique au matériel, avec notamment un système de δ -cycles pour simuler les dépendances entre signaux synchrones. Une solution serait d'inclure au programme vérifié la spécification du parallélisme SystemC sous forme d'ordonnanceur indéterministe représentant l'ensemble des comportements légaux. Cela augmenterait malheureusement la taille du programme à vérifier, et nuirait inévitablement à l'efficacité de l'outil.

Il existe aussi des choses intéressantes pour le langage Java, notamment l'outil JAVA PATHFINDER [HPVB00] qui traite à la fois la concurrence et les structures de haut-niveau. Une adaptation de cet outil à C++ et plus particulièrement à SystemC serait certainement utile pour la vérification des modèles TLM.

3.1.2 La chaîne d'outil LUSY

Pour le moment, la seule approche existante pour la vérification de modèles SystemC-TLM est donc celle de la chaîne d'outils LUSY (cf : [MMMC05a, MMMC06, Moy05]). Cet outil a été développé dans la même équipe, lors du doctorat de Matthieu Moy (2002-2005). Il s'agissait d'une collaboration entre STMicroelectronics et le laboratoire Verimag, comme pour les travaux de thèse présentés dans ce document.

Il ne s'agit pas d'un outil de vérification en soi, mais d'un traducteur de modèles SystemC-TLM vers un langage formel HPIOM, à partir duquel peuvent travailler des outils de vérifications existants. Le langage HPIOM permet de décrire des automates, et fournit un *produit synchrone* pour les assembler. Le choix d'une sémantique synchrone est justifié par le fait qu'un programme asynchrone peut toujours se traduire en un programme synchrone, mais que l'inverse est faux [Mil83, HB02]. Ainsi, il est possible de tenir compte à la fois des aspects synchrones et asynchrones de SystemC.

Il existe actuellement deux back-ends : l'un qui traduit ce langage vers Lustre, et l'autre qui le traduit vers SMV [McM92a]. Le langage SMV permet d'utiliser l'outil du même nom. Le langage Lustre

est un point d'entrée vers les outils LESAR (utilisant des BDDs, [HLR92]), NBAC (interprétation abstraite, [Jea03]) et PROVER PLUG-INTM de SCADETM (basée sur un solveur SAT).

La chaîne d'outils LUSY a aussi un autre intérêt : en fournissant une traduction de SystemC vers un langage, elle fournit une *sémantique formelle* à SystemC. Des travaux similaires ont été réalisés pour la formalisation de la concurrence en Java, grâce à une traduction vers des *machines à états abstraits* [GSW00]. Pour SystemC, il y a eu deux travaux avec le même objectif mais aucun n'est vraiment complet : [HGR⁺01] concerne la version 1.0 de SystemC et ignore la non-préemptivité ; [Sal03] ne traite qu'un sous-ensemble du langage et conclut par du pseudo-code difficile à contrôler.

Jusqu'à présent, LUSY n'a permis de prouver des propriétés que sur des exemples de petite taille. Des projets sont en cours pour en améliorer les performances, notamment avec des techniques d'abstractions plus évoluées, et grâce à de la *vérification par composants* [SCCS05].

Un lecteur étourdi pourrait croire qu'un outil de ce type constitue un pas vers la création d'un outil de synthèse TLM, puisque ils ont en commun le langage d'entrée (systemC-TLM) et le formalisme de sortie (systèmes synchrones et flot de données). Il n'en est rien : LUSY abstrait des informations (notamment les algorithmes pour la transformation des données), alors qu'un outil de synthèse devrait en ajouter (comme la micro-architecture du décodeur, par exemple).

La vérification formelle via LUSY est cependant un objectif ambitieux. Malgré toutes les optimisations que nous pourrions lui apporter, il est évident qu'il y aura des études de cas industrielles trop grosses ou trop complexes pour que leur correction soit formellement prouvée. En conséquence, il est nécessaire de s'intéresser à d'autres approches, que nous espérons compatibles avec des programmes de plus grande taille.

3.2 Validation par simulations

Une alternative à la vérification formelle est la *validation par simulations*, ou *test*. L'objectif des simulations est d'une part de trouver des erreurs, et d'autre part de fournir des arguments concluant que le programme sous test est correct. En dehors de programmes avec un ensemble d'entrées fini, les simulations ne permettent pas de prouver formellement l'absence d'erreurs. Il s'agit donc d'une méthode moins ambitieuse, mais cette ambition moindre lui permet de bien mieux passer à l'échelle.

3.2.1 Environnement de tests

La figure 3.1, et les explications de cette sous-section, rappellent les concepts généraux (et bien connus) de la validation par le test.

Tous les environnements de tests contiennent les mêmes grands éléments. Au cœur se trouve le *système sous test* (SST ou SUT), auquel sont reliés un *générateur d'entrées* et un *contrôleur des sorties* (parfois appelé *oracle*). Un système de *mesure de couverture* permet d'estimer si le système sous test a été suffisamment testé. Lors de la mise en œuvre, des composants peuvent être fusionnés ou au contraire divisés, mais les concepts sont toujours présents.

Le *système sous test* (SST) peut généralement être vu comme une simple boîte qui reçoit des entrées et génère des sorties. D'autres éléments de l'environnement de test peuvent dépendre de l'implantation interne du SST. Dans ce cas, la méthode de test est dite *boîte blanche*, et sinon : *boîte noire*.

Le *générateur d'entrées* dépend de l'interface du SST et de la spécification. Un programme est généralement prévu pour fonctionner dans un *environnement* bien particulier, qui doit être décrit dans la spécification. Dans le cas des systèmes réactifs, il est fréquent que la correction des entrées dépende

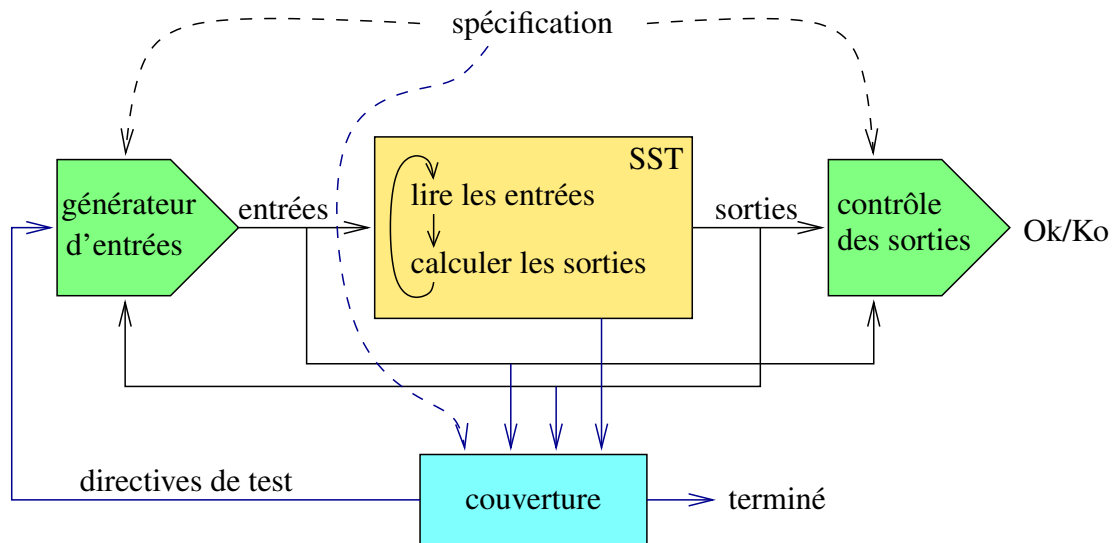


FIG. 3.1 – Architecture générale pour le test.

des sorties du SST. Considérons par un exemple un contrôleur d'ascenseur, dont les entrées sont constitués des boutons et d'un capteur de position, et dont les sorties commandent le déplacement de la cabine et l'ouverture des portes. Si le contrôleur, c'est-à-dire ici le SST, donne l'ordre de descendre, une position supérieure à la précédente serait incorrecte de la part du générateur. Le contrôleur d'ascenseur doit en revanche avoir un comportement sûr même si un passager presse les boutons de façon illogique.

La génération des entrées peut être :

- soit *manuelle* ; soit *automatique* ;
- soit *aléatoire* ; soit *dirigée*, en général par des informations sur la couverture ;
- soit *statique* (généralement en exécutant un modèle abstrait du SST) ; soit *dynamique*, c'est-à-dire en exécutant le SST lui-même.

L'*oracle*, dont le rôle est de vérifier que les sorties respectent la spécification, peut être séparé du SST, ou inclus dans le code source du SST sous forme d'*assertions*. Dans ce deuxième cas, l'oracle peut accéder à des informations internes au SST. En dehors des erreurs fatales (e.g. *segmentation fault* dans le cas du logiciel, ou blocage complet dans le cas du matériel), les oracles doivent prendre les entrées en compte pour détecter les comportements incorrects. Pour estimer si le résultat est correct, deux techniques sont employées :

- l'évaluation de *propriétés*, souvent écrites dans des *logiques temporelles* ;
- la comparaison à un *résultat de référence*, provenant d'un modèle plus abstrait, ou d'une version précédente et déjà validée du même SST.

Enfin, les *mesures de couverture* permettent d'estimer la fiabilité du SST en fonction des tests qui ont été exécutés avec succès. Il s'agit toujours d'estimations et non de garanties. Les mesures les plus employées utilisent les informations suivantes :

- la proportion de tests exécutés avec succès : plus un programme est fiable, plus il est difficile de le mettre en défaut ;
- les lignes de code source du SST exécutées au moins une fois (*couverture de code*) ;
- les lignes de code d'un modèle abstrait exécutées au moins une fois (*couverture de la spécification*) ;

- l'ensemble des fonctionnalités exercées, qui sont identifiées soit par des propositions atomiques, soit par des propriétés complexes (*couverture fonctionnelle*, [GHO⁺98]);
- la détection ou non d'erreurs dans des versions volontairement modifiées du SST (*injection d'erreurs*).

Considérons les deux programmes ci-dessous :

version A	version B
<pre>double f(double x) { return sin(1-x)/(1-x); }</pre>	<pre>double f(double x) { if (x==1) return cos(1); else return sin(1-x)/(1-x); }</pre>

Avec la simple exécution de l'appel `f(3.14)`, nous obtenons 100% de couverture de code pour la version A (incorrecte), et 50% pour la version B (correcte). Le critère de la couverture du code est donc insuffisant. Il en est de même pour les autres critères pris séparément. Une bonne analyse de couverture nécessite donc de combiner plusieurs critères.

A force d'écrire de nouveaux tests, le taux de couverture termine inévitablement à stagner. Cela peut avoir deux significations : soit que le programme est fiable, soit les tests générés ne sont pas assez pertinents et ne font que tester des cas déjà traités par les tests précédents. Certains critères peuvent avoir atteint leur maximum théorique (par exemple 100% pour le taux de couverture de code) mais en général il existe des critères dont le maximum effectif est inconnu.

La plupart des critères pour la mesure de couverture permettent d'identifier des *trous de couverture*. Les outils de génération de tests dirigée utilisent les informations sur ces trous de couverture, afin d'écrire de nouveaux tests qui ciblent ces trous. Cela permet de générer des tests plus pertinents, qui ont plus de chances de trouver des erreurs.

3.2.2 Outils pour la génération de tests

La validation étant très employée pour les descriptions RTL, toutes les grandes entreprises du domaine possèdent ou proposent des environnements de tests très complets avec un grand nombre de fonctionnalités périphériques, à l'exemple de X-Gen qui est développé par IBM [EJN⁺02]. Toujours pour le niveau RTL, la méthodologie présentée dans [SD02] permet de générer à la fois le générateur, l'oracle et la mesure de couverture à partir d'une spécification formalisée.

Au sein du laboratoire Verimag, nous disposons aussi d'un générateur de tests automatisé : l'outil Lurette [RNHW98]. Celui-ci génère des tests à partir d'un environnement formalisé sous la forme d'automates non-déterministes avec poids [RJR06]. Cet outil pourrait être utilisé sur les programmes Lustre générés par LUSSEY. [PHR04] décrit une méthode pour restreindre l'environnement en fonction des résultats des outils de vérification. Restreindre l'environnement revient en effet à guider la génération des tests. Il resterait ensuite le problème consistant à faire correspondre un contre-exemple sur le programme HPIOM à un contre-exemple sur le programme SystemC source.

Certains outils sont conçus pour permettre le lien entre des implantations VHDL et des modèles de référence écrit en SystemC. [FFP01] décrit l'architecture de la plateforme de test grâce à un langage baptisé IIR, à partir duquel ils peuvent générer soit du VHDL, soit du SystemC. Un système d'injection d'erreurs complète cet outil.

Les règles sur la propriété intellectuelle des composants matériels (IP) compliquent parfois les choses, puisque l'équipe de validation ne dispose alors que d'une vue incomplète de l'IP. [FFS01] décrit une méthode pour tester des modèles SystemC de composant à distance, c'est-à-dire avec des communications inter-composants passant par internet.

Dans cette thèse, nous nous limitons à la validation de modèles SystemC-TLM dont les sources nous sont disponibles. Plusieurs solutions industrielles existent pour générer des tests pour les modèles transactionnels, par exemple TestWizard, ou Specman qui utilise un langage de spécification baptisé *e*. [Bro02] proposait une comparaison de ces deux outils. Une étude de cas utilisant le langage *e* est disponible dans [KOW⁺01].

Une alternative à ces solutions propriétaires consiste à utiliser la librairie SCV (abréviation de *SystemC Verification*) [RS03]. Cette librairie a été utilisée pour valider des modèles TLM et leurs implantations RTL, développés par ARM [CLI⁺03]. Nous avons personnellement expérimenté cette librairie pour vérifier un petit modèle TLM. La procédure est la suivante :

1. nous commençons par définir une structure de donnée avec des champs numériques et d'autres champs de types énumérés ;
2. nous définissons des contraintes sur les champs :
 - simples : `addr.keep_only(0x1000, 0x1ffc)`
 - ou relationnelles : `SCV_CONSTRAINT(!(addr()==0x1004) || data()==1) ;`
3. nous fixons la valeur de certains attributs, déclarés comme n'étant pas aléatoires ;
4. nous demandons à la librairie de générer une solution, c'est-à-dire des valeurs pour chacun des autres champs ;
5. nous envoyons des données au système sous test à partir des données générées ;
6. nous attaquons le pas de simulation suivant, en retournant à l'étape 3.

À l'issue de cette expérimentation, nous avons pu tirer quelques enseignements. Tout d'abord, cette librairie apporte une vraie aide pour la génération de données, à condition que l'on puisse se limiter à des contraintes linéaires. Ensuite, il est aussi possible de gérer la *partie contrôle* avec cette librairie. En effet, la partie contrôle est généralement définie par un automate, or les automates peuvent facilement se coder sous forme de booléens représentant les états, et de contraintes représentant les transitions valides. Ce n'est cependant pas le plus simple, et il est préférable d'utiliser d'autres outils complémentaires pour cela.

Au début de cette thèse, nous avons commencé à développer des générateurs génériques pour les protocoles TLM utilisés à STMicroelectronics (dont le protocole TAC [tac05]). Ceux-ci permettaient de générer des tests manuellement et dynamiquement, puis de les rejouer. Ils offraient aussi la possibilité de coder des structures de contrôle simples, ainsi que des bases pour la génération aléatoire de données. Pour continuer dans cette voie, c'est-à-dire dans la génération automatique de tests, il serait nécessaire de fixer un langage pour les spécifications, comme pour les outils [KOW⁺01, SD02, RJR06] précédemment cités. L'absence d'un langage formel pour les spécifications obligerait à écrire manuellement chacun des éléments de l'environnement de test (générateur, oracle et couverture), et cela ne permettrait pas une productivité satisfaisante.

3.3 Combinaison des méthodes

La vérification formelle a l'avantage de l'exhaustivité, mais a pour défauts, d'une part d'être très sensible à la taille du programme à vérifier, d'autre part d'échouer souvent à fournir un vrai contre-exemple. Les avantages et inconvénients du test sont inverses, d'où l'idée d'essayer de combiner ces deux méthodes pour profiter au mieux de leurs qualités respectives. Il est bien sûr possible d'utiliser plusieurs méthodes pour la validation d'un même système [BGB02], mais il est bien plus intéressant de les combiner en une seule et même technique. C'est ce dont nous allons parler dans cette section.

3.3.1 Génération de tests ciblant un trou de couverture

La génération aléatoire, ou dirigée par des méthodes simples, permet d'atteindre rapidement un bon taux de couverture, mais les derniers trous dans la couverture demandent souvent beaucoup de travail pour être comblés. Une idée, désormais très répandue, consiste à utiliser un outil de vérification pour générer un test ciblant ce trou [GFL⁺96].

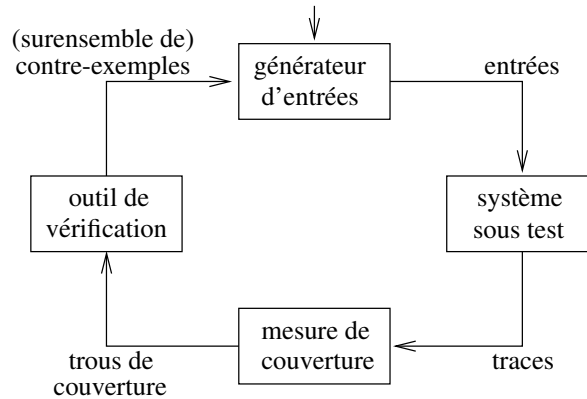


FIG. 3.2 – Principe de la génération de tests dirigée par la couverture.

Le principe est résumé par la figure 3.2. De premiers tests couvrent partiellement les objectifs. L'un des objectifs non-couverts est extrait puis codé sous la forme d'un *pseudo état d'erreur*. L'outil de vérification est lancé à la recherche de cette pseudo erreur. Le résultat attendu est un contre-exemple menant à la pseudo erreur, et donc couvrant l'objectif ciblé. Cette procédure est itérée jusqu'à obtenir une couverture complète.

La théorie est jolie mais il y a un problème pratique majeur : le test est généralement utilisé lorsque le programme est trop complexe pour passer dans les outils de vérification. Par conséquent, l'exécution naïve de l'outil de vérification pour trouver la pseudo erreur a toutes les chances d'échouer ; d'autant plus que l'utilisation d'abstraction serait nuisible à l'obtention d'un véritable contre-exemple. Plusieurs idées ont été expérimentées et publiées pour résoudre ce problème.

- L'outil décrit dans [HSH⁺00] commence par lancer une simulation jusqu'à un état supposé proche d'un état non-couvert. L'outil de vérification est lancé seulement à partir de ce point, et a donc moins de travail pour ajouter un nouvel état à la couverture. Il est ensuite possible de relancer des simulations à partir de ce nouvel état.
- Dans [Ip00], un historique abstrait des explorations effectuées est maintenu. Grâce à cette historique, il est ensuite possible de transformer d'anciens stimulus de test pour en obtenir de nouveaux, accédant à des espaces encore non-couverts.
- [FZ03] reprend une idée semblable, mais utilise des réseaux bayésiens pour accumuler des connaissances sur le comportement du SST.

Utiliser cette technique pour la validation de modèles SystemC-TLM nécessiterait un outil de vérification formelle pour ce même langage. Nous disposons d'ores et déjà de la chaîne d'outils LUSSY, mais elle doit encore être améliorée sur deux points pour être utilisable dans ce contexte : d'une part sur les performances (par exemple grâce à la vérification par composants), d'autre part sur la génération automatique d'un contre-exemple SystemC (par exemple en gardant une trace de la correspondance SystemC - HPIOM, dans l'idée de la "*refinement map*" de [TYB03]).

A défaut de pouvoir appliquer de la vérification formelle sur le système à valider, que ce soit à cause d'un problème de langage ou d'une trop grande complexité, il est possible de générer des tests

couvrant intégralement un modèle abstrait et exécutable du SST, puis de raffiner ces tests (abstrait) pour obtenir des tests concrets couvrant efficacement le système sous test. [BBBD02] présente un outil permettant de générer des tests pour des systèmes sur puce à partir de modèles abstraits écrits manuellement sous forme d'automates finis.

3.3.2 Vérification à la volée et simulations équivalentes

Une autre idée consiste, non seulement à observer le comportement du SST sur les entrées courantes, mais aussi à essayer de détecter des erreurs pour des entrées voisines. Connaissant une partie du fonctionnement interne du SST, l'impact des variations d'un type d'entrées est calculé à la volée. Si cet impact peut causer une erreur, alors un avertissement est émis. Cette technique est connue en anglais sous le nom de *runtime verification*.

Dans [SBN⁺97], des accès mal synchronisés à une donnée peuvent être détectés, même s'il n'apparaissent qu'avec un *ordonnancement* différent, grâce une observation des prises de verrous par les différents processus. La vérification à la volée, combinée à de la génération dirigée utilisant l'outil de vérification JAVA PATHFINDER [HPVB00], a été utilisée avec succès pour la vérification d'un contrôleur d'inclinaison de la NASA [ABG⁺05].

Déjà dans [GG75], plusieurs critères sont définis pour qualifier qu'un jeu de tests est suffisant pour trouver toutes les éventuelles erreurs. Prouver que toutes les erreurs sont détectées sans avoir essayé toutes les entrées possibles repose sur le fait suivant : deux entrées différentes peuvent donner des résultats suffisamment *liés* pour que la correction avec le premier jeu d'entrées soit *équivalente* à la correction avec le deuxième jeu d'entrées.

Cela n'est pas applicable pour tous les types d'entrées. En effet, il faut disposer d'un moyen mathématique pour estimer les conséquences d'une modification sur une entrée sans devoir l'appliquer. La solution consiste à profiter de cette technique pour les entrées qui s'y prêtent, et utiliser les méthodes classiques pour les autres entrées.

Au début de ces travaux de thèse, après quelques pistes non poursuivies (sous-sections 3.2.2 et 3.3.1), nous nous sommes intéressés à ces techniques de *runtime verification* pour résoudre le problème de l'ordonnancement en SystemC. De là, nous avons rapidement évolué vers la *réduction d'ordre partiel dynamique* [FG05], qui étendent le concept décrit ci-dessus, avec des techniques pour générer itérativement des tests voisins mais non équivalents.

Chapitre 4

Le problème de l'ordonnancement

Sommaire

4.1	L'ordonnancement en SystemC	44
4.1.1	Algorithme de l'ordonnanceur	44
4.1.2	Les actions de communication	46
4.1.3	Ajout : l'instruction "yield"	46
4.2	Exemple	47
4.2.1	Exemple avec deux processus	47
4.2.2	Version étendue à trois processus	48
4.3	Conséquences pour des cas réels	50
4.3.1	Blocage au démarrage	50
4.3.2	Procédure d'arbitrage	50
4.4	Réalisation de systèmes indépendants de l'ordonnancement	52
4.4.1	Exemple : système d'arbitrage indépendant de l'ordonnancement	52
4.4.2	Analyse de l'exemple et limitations	53
4.4.3	Conclusion	54

Nous simulons des modèles de systèmes concurrents sur des machines séquentielles. Cela impose d'ordonner les processus du modèle simulé afin de les exécuter un à un, c'est-à-dire déterminer à quel processus donner la main, et définir quand la reprendre. Certains langages, comme SystemC, n'ordonnent pas les processus de façon déterministe, permettant ainsi plusieurs exécutions différentes. Comme nous allons le voir, ceci a des conséquences importantes sur la conception et la validation des modèles. Le rôle de ce chapitre est d'explicitier les raisons qui nous ont amenés à nous pencher sur la génération automatique d'ordonnements.

Nous allons tout d'abord expliquer la politique d'ordonnements de SystemC en ciblant sur les aspects qui nous concernent (section 4.1). Un premier exemple viendra illustrer cette section et introduire le problème posé par l'indéterminisme de l'ordonneur SystemC (section 4.2). Des problèmes dus à l'ordonnement se posent dans des programmes conçus au sein de STMicroelectronics ; nous détaillerons les cas les plus fréquents (section 4.3). Enfin, nous étudierons ce qui aurait pu être une façon de contourner le problème (section 4.4).

4.1 L'ordonnancement en SystemC

Les deux principales caractéristiques de l'ordonnanceur SystemC sont d'être *non-préemptif*, c'est-à-dire que ce sont les processus qui décident quand rendre la main, et *localement asynchrone*. La non-préemptivité a deux grands avantages. D'une part l'écriture des modèles est facilitée car le comportement à l'exécution devient alors plus prévisible. D'autre part, cela permet de limiter au strict minimum les changements de contexte, c'est-à-dire le passage d'un processus à un autre, qui est une opération coûteuse en temps. Le principal inconvénient est qu'un processus peut bloquer tout le programme s'il boucle indéfiniment sans rendre la main. Pour cette raison, les ordonnanceurs des systèmes d'exploitation sont souvent préemptifs.

Le simulateur SystemC gère une horloge virtuelle dont la valeur avance en fonction des instructions présentes dans le programme. Nous appelons *temps simulé* la valeur de cette horloge, par opposition au temps réel qui s'écoule pendant la simulation. Cette horloge virtuelle évolue de façon discrète. Entre deux avancements successifs, les processus du programme s'exécutent de façon *asynchrone*.

4.1.1 Algorithme de l'ordonnanceur

La spécification du langage SystemC est donnée dans un document nommé "Language Reference Manual (LRM)" [Ope03]. Il s'agit d'un document essentiellement textuel. Pour l'ordonnancement, un pseudo-code d'ordonnanceur est fourni dans les annexes. L'indéterminisme de l'ordonnanceur est clairement signalé dans ce document et ne résulte pas d'une ambiguïté involontaire dans sa définition.

L'OSCI fournit aussi une implantation, mais celle-ci n'a pas de valeur normative. Elle mérite cependant d'être étudiée. D'une part il s'agit de l'implantation la plus utilisée. D'autre part, elle est distribuée sous une licence de type "libre" ce qui permet de l'instrumenter, la compléter ou la modifier à notre guise.

Comme vu au chapitre 2, un modèle se compose de plusieurs composants reliés par des canaux de communications. Dans chaque composant et canal, se trouve initialement un nombre quelconque de processus. Les processus peuvent *migrer* de composants via le mécanisme des transactions, c'est-à-dire qu'ils peuvent exécuter du code de plusieurs modules différents. Les processus communiquent entre eux par des variables partagées et des événements SystemC `sc_event`, ainsi que par des structures de plus haut niveau comme les signaux SystemC `sc_signal`.

Il existe trois types de processus en SystemC : les `SC_THREAD` qui sont la forme la plus générale, mais aussi les `SC_METHOD` et les `SC_CTHREAD`. Les `SC_CTHREAD` sont synchronisés avec une horloge et ne sont donc pas utilisés dans les modèles transactionnels qui nous intéressent dans cette thèse. Les `SC_METHOD` sont des processus très simples qui ont la particularité d'avoir une pile d'exécution vide entre deux pas d'exécution. Ils pourraient systématiquement être écrits sous la forme de `SC_THREAD` mais les `SC_METHOD` permettent une implantation plus efficace. En effet, pour un pas d'exécution d'une `SC_METHOD`, un appel de fonction suffit, alors que pour un `SC_THREAD` un changement de contexte est nécessaire (enregistrement de l'ancienne pile d'exécution et restauration de la nouvelle).

La première étape d'une simulation consiste à construire l'architecture statique du modèle SystemC, en créant les composants et établissant les connexions. Il s'agit de la *phase d'élaboration* (ELAB). Ensuite, l'ordonnanceur lance l'exécution proprement dite du modèle, en exécutant les processus selon l'automate informel de la figure 4.2. Une exécution d'un modèle SystemC consiste en une séquence de *phases d'évaluations* (EV), séparées par des *phases de mises à jour des signaux* (UP), et des *avancements du temps simulé* (TE), comme décrit par la figure 4.1.

Toujours selon sa spécification, un ordonnanceur SystemC doit se comporter de la façon suivante :

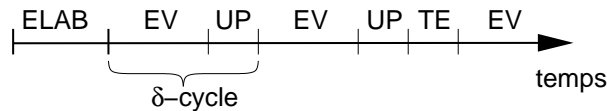


FIG. 4.1 – Diagramme d'une exécution

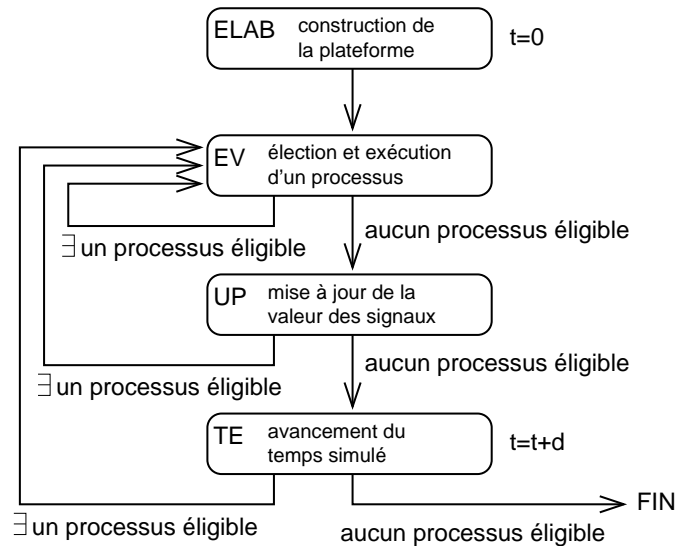


FIG. 4.2 – Automate de l'ordonnanceur SystemC

ELAB A la fin de la phase d'élaboration, certains processus sont *éligibles*, les autres sont *en attente*.

EV Pendant la phase d'évaluation, les processus éligibles sont exécutés dans un **ordre non-spécifié**, de façon non-préemptive, et se suspendent d'eux-mêmes quand ils rencontrent une instruction d'attente *wait*. Il y a deux types d'instructions d'attente : un processus peut attendre sur du temps en spécifiant une durée, ou attendre la notification d'un ou plusieurs événements SystemC¹. Pendant qu'un processus s'exécute, il peut accéder à des variables partagées ou des signaux, rendre éligible d'autres processus en notifiant un événement, ou programmer des *notifications retardées*. Un processus éligible le reste tant qu'il n'a pas été exécuté.

UP Quand il ne reste plus de processus éligible, la valeur des signaux SystemC est mise à jour et les notifications retardées d'un δ -cycle sont effectuées. Cela peut rendre éligibles des processus. Un δ -cycle est la durée entre deux phases de mise à jour consécutives. L'ordre des opérations durant cette phase est sans conséquence puisqu'il n'y a pas d'interactions entre les différents processus. Les accès retardés multiples à un même objet sont traités par des règles ad hoc pendant les phases d'évaluation.

TE Quand il n'y a aucun nouveau processus éligible à la fin d'une phase de mise à jour, l'ordonnanceur laisse le temps s'écouler jusqu'au réveil des processus ayant la durée d'attente restante la plus courte.

La notification d'un événement SystemC peut être immédiate, retardée d'un δ -cycle ou retardée d'une autre durée spécifiée. Les processus peuvent ainsi devenir éligibles à n'importe laquelle des

¹ Il existe aussi un troisième type mixte d'instruction d'attente qui consiste à attendre un événement pendant une durée maximale fixée. Passé ce délai, le processus redeviendra obligatoirement éligible. Ce type d'instruction est absente de toutes les études de cas qui nous ont été fournies et nous n'en parlerons pas dans cette thèse.

trois phases EV, UP ou TE. Aucune des trois boucles de l'automate de l'ordonnanceur n'est bornée en nombre d'itérations.

Il est important de noter que les événements SystemC ne sont pas persistants. C'est-à-dire que lors de la notification effective d'un événement, seul les processus actuellement en attente sur lui sont concernés. Un processus ne peut pas savoir directement si un événement a été notifié dans le passé ou non.

4.1.2 Les actions de communication

On appelle *objet partagé* tout objet qui est accessible depuis au moins deux processus, et *action de communication* toute action qui modifie ou utilise un objet partagé. La librairie SystemC fournit un grand nombre de structures permettant aux processus de communiquer. Cependant, la plupart peuvent être définies à partir d'autres structures de plus bas niveau. Pour les communications internes à une phase d'évaluation, seul deux types d'objets partagés sont à considérer : les événements et les variables.

Il y a deux opérations sur les événements : `wait` et `notify`. Dans certains cas, il peut être utile de distinguer les notifications *réussies* des notifications *manquées* selon qu'elles ont ou non rendu un processus éligible.

4.1.3 Ajout : l'instruction "yield"

En SystemC, le délai correspondant à un δ -cycle se note `SC_ZERO_TIME`. Il est possible d'écrire dans un programme une instruction `wait (SC_ZERO_TIME)`, ou encore `wait (0, SC_NS)` qui est strictement équivalente. Malheureusement, ces deux instructions sont trompeuses. Selon la spécification de l'ordonnanceur, tous les processus changent de delta-cycle en même temps. Par conséquent, un processus qui exécute deux fois `wait (0, SC_NS)` se réveillera toujours après un processus qui ne l'exécute qu'une fois. Mathématiquement, on a $SC_ZERO_TIME + SC_ZERO_TIME > SC_ZERO_TIME$, ce qui implique de considérer que `SC_ZERO_TIME` correspond à une durée non nulle. Il est possible de synchroniser des processus en comptant les δ -cycles comme on le ferait avec une horloge globale, mais ce ne serait pas considéré comme une façon propre de programmer au niveau TLM.

Dans certains cas, on peut souhaiter rendre la main à l'ordonnanceur afin que d'autres processus puissent s'exécuter mais sans pour autant se synchroniser avec l'ensemble des autres processus. Pour cela, il faudrait disposer d'une instruction qui permet de rendre la main mais en redevenant immédiatement éligible. On nommera cette instruction `yield`.

L'absence de cette instruction dans la spécification SystemC peut être contournée en ajoutant un processus jouant le rôle de réveil. Le principe consiste à demander au processus réveil d'émettre un événement de façon immédiate, comme montré par la figure 4.3. En procédant ainsi, l'ensemble des exécutions possibles est équivalent à celui obtenu grâce à une instruction `yield`, pour peu que l'on masque le processus supplémentaire.

Étant donné que ce que l'on souhaite faire est compliqué mais possible, ajouter cette fonction au noyau de SystemC se justifie. C'est ce que nous avons fait pour le cas des processus `SC_THREAD`.

Dans la suite, cette fonction nous servira pour écrire des modèles strictement non-temporisés. De plus, nous en aurons besoin à la sous-section 7.3.4 pour un problème de modélisation mathématique. Par ailleurs, cette fonction pourrait être utilisée pour simuler de la préemptivité, et donc améliorer la représentativité des modèles par rapport au système réel. Bien sûr, cette fonction n'a une utilité que si l'on s'intéresse à explorer l'espace des ordonnancements.

```

sc_event e, f;
void top::Compute() {
    cout <<"before\n";
    e.notify();
    wait(f);
    cout <<"after\n";
}

void top::waker() {
    while (true) {
        wait(e);
        f.notify();
    }
}

```

FIG. 4.3 – Utilisation d'un processus réveil pour simuler une instruction `yield`

4.2 Exemple

Nous allons maintenant examiner en détail un exemple et sa variante. Il s'agit d'illustrer la communication entre processus via des variables partagées et des notifications immédiates d'événements. Cet exemple reproduit sous une forme réduite les problèmes qui peuvent apparaître à cause de l'indéterminisme de l'ordonnancement, sur un programme de taille industrielle.

4.2.1 Exemple avec deux processus

L'exemple `bozo` se compose d'un module contenant deux processus `P` et `Q`. Ces deux processus communiquent en utilisant un événement `e` et une variable `x` valant initialement 0. Le code des processus est fourni par la figure 4.4.

```

void top::P() {
    wait(e);
    wait(20, SC_NS);
    if (x) cout << "Ok\n";
    else cout << "Ko\n";
}

void top::Q() {
    e.notify();
    x = 0;
    wait(20, SC_NS);
    x = 1;
}

```

FIG. 4.4 – Exemple `bozo`

Les trois tableaux qui suivent représentent l'ensemble des exécutions possibles de cet exemple. La première correspond à ce que l'on obtient avec l'ordonnancement OSCI, sous réserve que le processus `Q` soit déclaré après `P`, et qu'il n'y en ait pas d'autres.

éligibles	choisi	actions
P; Q	P	<code>wait(e);</code>
Q	Q	<code>e.notify(); x = 0; wait(20, SC_NS);</code>
P	P	<code>wait(20, SC_NS);</code>
∅	UP	<i>aucune mise à jour</i>
∅	TE	<i>écoulement de 20 ns</i>
P; Q	Q	<code>x = 1;</code>
P	P	<code>cout << "Ok\n";</code>
∅		<i>fin de la simulation</i>

TAB. 4.1 – ordonnancement PQP||QP

éligibles	choisi	actions
P; Q	P	<code>wait(e);</code>
Q	Q	<code>e.notify(); x = 0; wait(20, SC_NS);</code>
P	P	<code>wait(20, SC_NS);</code>
∅	UP	<i>aucune mise à jour</i>
∅	TE	<i>écoulement de 20 ns</i>
P; Q	P	<code>cout << "Ko\n";</code>
Q	Q	<code>x = 1;</code>
∅		<i>fin de la simulation</i>

TAB. 4.2 – ordonnancement PQP||PQ

éligibles	choisi	actions
P; Q	Q	<code>e.notify(); x = 0; wait(20, SC_NS);</code>
P	P	<code>wait(e);</code>
∅	UP	<i>aucune mise à jour</i>
∅	TE	<i>écoulement de 20 ns</i>
Q	Q	<code>x = 1;</code>
∅		<i>fin de la simulation</i>

TAB. 4.3 – ordonnancement QP||Q

Les deux premières simulations (tableaux 4.1 et 4.2) diffèrent uniquement par l'ordre des deux derniers choix. La valeur de x lue pour tester la condition diffère selon l'ordonnancement. Dans la littérature anglophone, cela est appelé *data-race*. Cela amène à un comportement différent, dans le premier cas "Ok" est affiché, dans l'autre on obtient "Ko". Pour un exemple réel, on aurait pu obtenir le résultat attendu dans un premier cas, et un résultat incorrect dans l'autre.

La troisième simulation, décrite par le tableau 4.3, se distingue des deux autres par le fait que l'instruction `wait(e)` est exécutée après la notification `e.notify()`. La conséquence est que le processus P rate la notification et reste bloqué sur cette instruction jusqu'à la fin de la simulation. Des exemples réels peuvent se bloquer partiellement ou intégralement à cause de ce type de synchronisation défectueuse. Cela est une forme d'interblocage ("*deadlock*").

Dans le cas présent, nous obtenons un blocage lorsque l'événement notifié n'est reçu par aucun autre processus. Cependant, lors d'exécutions de modèles réels, il est fréquent de trouver des notifications reçues par personne sans que cela ne corresponde à une erreur.

4.2.2 Version étendue à trois processus

L'exemple `bozo++`, présenté figure 4.5, est une extension simple de l'exemple `bozo`. La seule modification consiste en l'ajout d'un processus R, déclaré en premier et qui ne communique pas directement avec les autres processus. On ne décrit qu'une exécution : celle obtenue avec l'ordonnateur de l'implantation OSCI (tableau 4.4).

La première constatation concerne l'ordonnateur OSCI. Le comportement du sous-système composé de P et Q a été modifié par la présence de R, bien que celui-ci ne communique pas directement avec le reste du système.

La cause de la perturbation créée par R se trouve dans la structure de données utilisée par l'ordonnateur pour stocker les attentes sur du temps. Sémantiquement, on a besoin d'une file à priorités. La

4.2. Exemple

```

void top::R() {
    sc_time T(20, SC_NS);
    wait(T);
}
void top::P() {
    comme dans l'exemple bozo
}
void top::Q() {
    comme dans l'exemple bozo
}

```

FIG. 4.5 – Exemple bozo++

éligibles	choisi	actions
P; Q; R	R	wait(20, SC_NS);
P; Q	P	wait(e);
Q	Q	e.notify(); x = 0; wait(20, SC_NS);
P	P	wait(20, SC_NS);
∅	UP	<i>aucune mise à jour</i>
∅	TE	<i>écoulement de 20 ns</i>
P; Q; R	R	
P; Q	P	cout << "Ko\n";
Q	Q	x = 1;
∅		<i>fin de la simulation</i>

TAB. 4.4 – ordonnancement OSCI RPQP||RPQ

priorité est déterminée par l'heure de réveil, c'est-à-dire l'heure courante plus le délai donné en argument. En effet, les heures de réveil à stocker arrivent dans un ordre quelconque, et on retire toujours l'heure de réveil la plus petite. L'implantation OSCI utilise un *tas*² pour représenter cette file à priorité. Cette structure de données a l'avantage de l'efficacité ($\log(n)$ pour l'ajout et pour le retrait), mais a l'inconvénient d'être instable : l'ajout d'un élément perturbe l'ordre des éléments déjà présents.

En conséquence de cette instabilité, l'ordonnancement généré par l'implantation OSCI est quasiment impossible à prévoir, et le comportement d'un sous-système est dépendant de l'ensemble du système. Le tableau 4.5 donne un aperçu de ce qui peut se passer quand on change l'ordre de déclaration des processus, ou que l'on change simplement la valeur d'un délai. Notre conclusion est que même un programmeur n'utilisant que l'implantation OSCI doit considérer l'ordonnancement comme étant partiellement aléatoire.

	T < 20ns	T = 20ns	T > 20ns
R déclaré avant P, Q	Ko	Ko	Ok
R déclaré après P, Q	Ko	Ok	Ok

TAB. 4.5 – Résultat obtenu pour bozo++ en fonction de T

On considère maintenant que l'on dispose d'un ordonnanceur interactif permettant d'essayer plusieurs ordonnancements. L'exemple bozo acceptait trois exécutions différentes qui chacune menait à un résultat distinct. L'exemple bozo++ accepte 30 ordonnancements différents mais il n'y a toujours que 3 résultats distincts, à savoir "Ok", "Ko" ou blocage. Intuitivement, les instants auxquels on choisit d'exécuter les deux occurrences de R n'ont pas de conséquence. Il y a 12 ordonnancements menant au résultat "Ok", autant menant à "Ko" et 6 menant à un blocage. L'important pour la validation est d'essayer suffisamment d'ordonnancements pour obtenir ces trois résultats, mais nous voyons déjà sur cet exemple qu'il n'est pas nécessaire d'exécuter tous les ordonnancements possibles.

²Un *tas* est un arbre binaire complet tel que chaque noeud soit plus petit que ses fils.

4.3 Conséquences pour des cas réels

Nous présentons dans cette section les erreurs de synchronisation et les manques de déterminisme que l'on trouve le plus fréquemment dans les programmes industriels, et qui ne se révèlent que pour certains ordonnancements.

4.3.1 Blocage au démarrage

Lors du démarrage d'un modèle, il est courant que les composants cibles doivent exécuter une phase d'initialisation avant d'être prêts à répondre à des communications externes. Il arrive malheureusement que les composants initiateurs fassent des requêtes avant que les composants cibles soient prêts.

processus initiateur i	processus cible c
<pre> <i>composant initiateur I :</i> void I::compute() { port.write(@T+@instr, data); port.write(@T+@start, 1); } <i>composant cible T :</i> void T::write(addr, data) { if (addr==@start) start_work.notify(); else if ... } </pre>	<pre> <i>composant cible T :</i> void T::compute() { read_config_file(filename); while(true) { wait(start_work); proceed(); } } </pre>

FIG. 4.6 – Programme avec blocage éventuel lors du démarrage.

La figure 4.6 fournit un exemple de programme ayant ce problème. Si le processus cible c est élu en premier, alors tout se passe comme voulu. Par contre, si le processus initiateur i est élu en premier, l'événement `start_work` sera notifié avant d'être attendu, puisque le processus i ne rend pas la main avant ce point. En conséquence, le programme se bloque puisque le composant cible n'attaque jamais le traitement de la requête.

Il y a deux types de solution pour ce problème :

- en modifiant l'initiateur : si on ajoute une attente sur du temps suffisamment longue, le processus cible aura le temps de s'initialiser ;
- en modifiant la cible : il est possible de simuler un *événement persistant* en ajoutant une variable partagée, puis en utilisant pour la notification et l'attente les bouts de code donnés par la figure 4.7.

La deuxième solution a l'avantage d'être purement locale, contrairement à la première qui peut avoir des conséquences globales sur les synchronisations (le processus initiateur se retrouve retardé par rapport à la version originale).

4.3.2 Procédure d'arbitrage

Il arrive que plusieurs processus souhaitent accéder en même temps à une même ressource, qui n'accepte qu'un seul accès à la fois. Le problème, classique, est de réaliser un bout de programme permettant de séquentialiser les requêtes. Nous avons d'abord rencontré ce problème dans le cadre des

Notification	Attente
<pre>e.notify(); x = true;</pre>	<pre>if (!x) { wait(e); } x = false; //reset</pre>

FIG. 4.7 – Modélisation d'un événement persistant avec un couple booléen - événement (x , e). La variable permet de mémoriser la notification au cas où elle ait eu lieu avant d'être attendue.

composants bus `tac_seq` et `tac_arbiter`. Ces bus reçoivent plusieurs transactions, et doivent les transmettre aux composants cibles sans que ces transactions se chevauchent. Désormais, seul le bus `tac_router` est utilisé pour les modèles purement fonctionnels. Ce bus, très abstrait, transmet directement les transactions reçues sans tenir compte des autres. Le même problème se retrouve cependant dans d'autres composants encore en usage, comme par exemple les contrôleurs d'interruptions. De plus, des bus similaires sont et seront toujours utilisés pour les niveaux d'abstraction inférieurs.

Une première implantation possible pour un contrôleur d'interruptions (ITC) est donnée par la figure 4.8, sous une forme très épurée. Celle-ci est basée sur un verrou SystemC `sc_mutex`. Les interruptions sont transmises par le protocole `tlm_synchro`, fourni avec le protocole TAC [tac05].

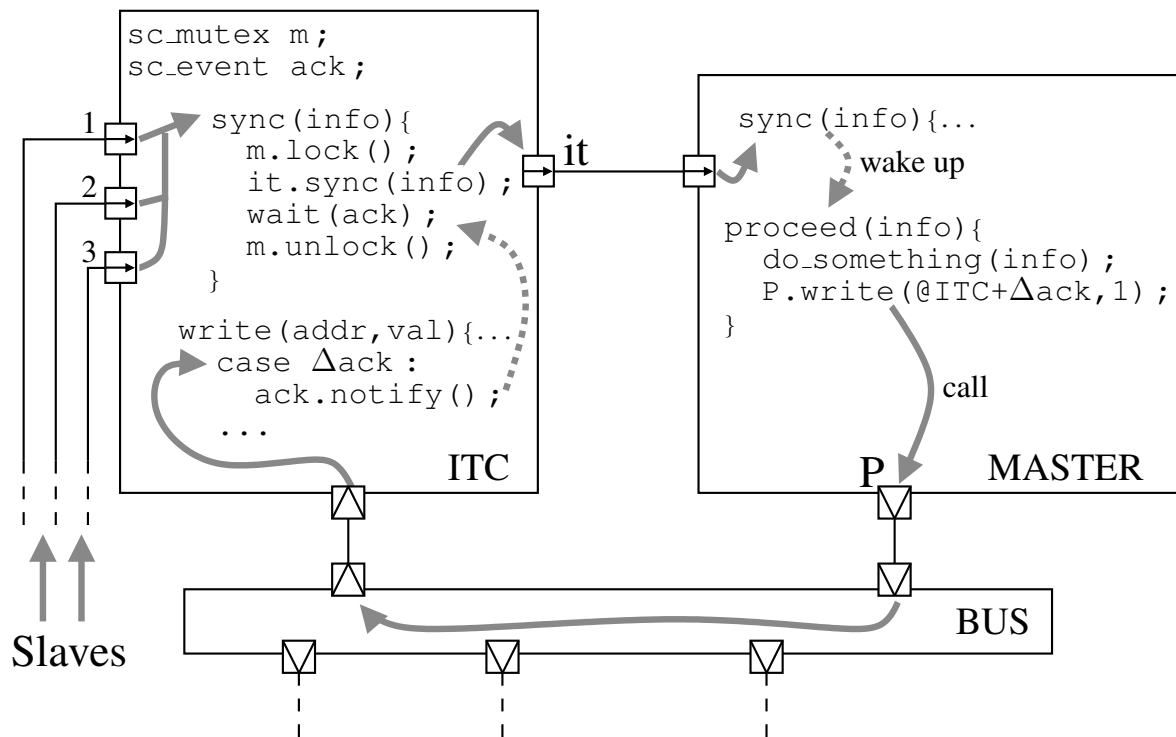


FIG. 4.8 – Contrôleur d'interruptions basé sur un verrou SystemC basique.

Lorsqu'une interruption arrive, la fonction `sync` est appelée par le processus du composant esclave. Le paramètre `info` contient en général une donnée fournie par le processus esclave, ainsi que l'identifiant du port. Plusieurs processus esclaves peuvent appeler la fonction `sync` en même temps. La fonction `proceed` est ici une méthode SystemC (`SC_METHOD`) sensible à l'arrivée d'une interruption ; cette fonction n'est pas directement appelée par le processus d'un autre composant. Le verrou SystemC `m` garantit que le composant maître ne recevra pas une nouvelle interruption tant qu'il n'aura

pas validé la précédente.

Il ne s'agit pas de la solution utilisée à STMicroelectronics (qui n'est pas publique). Cette solution a en effet deux problèmes :

1. l'ordre de traitement des interruptions dépend directement de l'ordonnancement, et est donc imprévisible ;
2. lorsque n processus sont simultanément en attente du verrou, l'implantation de la classe `sc_mutex` n'est pas efficace car elle provoque $n - 1$ changements de contexte inutiles lors de chaque appel à la fonction `unlock`.

La solution initiale de STMicroelectronics est censée résoudre ces deux points grâce à un stockage explicite des requêtes en attente, et un tri en fonction des priorités. Ces priorités sont généralement déterminées par le numéro de port d'où provient l'interruption. En étudiant de plus près cette solution (à la main), nous avons constaté que le premier point n'était pas tout à fait résolu. En effet, dans certains cas, l'ordre de traitement dépend toujours de l'ordonnancement : par exemple lorsque une requête arrive au δ -cycle qui suit l'arrivée d'une requête moins prioritaire. Des ingénieurs nous ont dit avoir déjà rencontré ce cas, et confirmé la présence du problème.

Le fait que l'ordre de traitement dépende de l'ordonnancement n'est pas un bug en soi. Le modèle peut être fonctionnellement correct quel que soit l'ordre de traitement des interruptions. Cependant, cela complexifie la validation du modèle car l'ordre de traitement des requêtes peut conditionner l'apparition d'une erreur de synchronisation. Exécuter une seule fois un test est dans ce cas insuffisant pour montrer qu'il passera toujours. Autrement dit, même un très bon taux de couverture du code source n'apporte aucune garantie si tous les ordonnancements possibles n'ont pas été considérés.

4.4 Réalisation de systèmes indépendants de l'ordonnancement

Une solution, pour éviter de devoir simuler chaque test avec plusieurs ordonnancements différents, serait de n'écrire que des programmes indépendants de l'ordonnancement. Un programme peut être dit indépendant de l'ordonnancement si, pour des données fixées, il est, soit correct pour tous les ordonnancements possibles, soit incorrect pour tous les ordonnancements possibles. Le bénéfice majeur serait une validation grandement facilitée, mais cela pose aussi de nombreuses questions comme : sait-on écrire des programmes indépendants de l'ordonnancement ? Comment prouver par construction qu'ils sont bien indépendants ? Est-il toujours souhaitable d'être indépendant à l'ordonnancement ?

4.4.1 Exemple : système d'arbitrage indépendant de l'ordonnancement

Nous allons montrer ici comment construire une version indépendante de l'ordonnancement du contrôleur d'interruptions décrit par la figure 4.8. Nous avons vu que l'indéterminisme provenait de l'objet `sc_mutex`. L'idée est de garder la même structure générale, mais de remplacer le verrou SystemC par un objet d'une nouvelle classe `tlm_arbiter`. Cette nouvelle classe a les deux mêmes méthodes publiques : `lock` et `unlock`, mais la fonction `lock` prend désormais une priorité en argument.

Une requête est traitée chaque fois que la méthode `lock` retourne. L'objectif est que l'ordre de traitements des requêtes ne dépende pas de l'ordonnancement des processus. Par ailleurs, pour des raisons d'efficacité, un processus doit être mis en attente au maximum une fois ; toute mise en attente supplémentaire causerait un changement de contexte inutile et coûteux.

Pour trier les requêtes, nous disposons de deux critères : la priorité fournie en argument et la chronologie. L'ordonnancement a généralement des conséquences sur cette chronologie. Ici, nous faisons

l'hypothèse suivante : le δ -cycle où arrive une requête est fixe, mais l'ordre des requêtes arrivant durant un même δ -cycle peut varier selon l'ordonnancement. Par conséquent, les requêtes d'un même δ -cycle doivent être triées selon leur priorité. De plus, une requête doit patienter au minimum un δ -cycle, puisque jusqu'à la fin de son δ -cycle d'arrivée elle peut être doublée par une requête prioritaire.

Notre implantation de la classe `tlm_arbiter` utilise deux listes :

- `pending_requests` : liste des requêtes prêtes à être traitées ; elles peuvent être triées soit uniquement selon leur priorité (comportement du `tac_arbiter`), soit en tenant compte d'abord de la chronologie (comportement semblable au `tac_seq`) ;
- `new_requests` : liste des requêtes arrivées durant le δ -cycle courant, triées selon leur priorité.

Le contenu de `new_requests` doit être transféré dans `pending_requests` lors du passage au δ -cycle suivant. Pour cela nous utilisons le mécanisme de mises à jour synchrones de SystemC, qui est utilisé notamment pour les classes `sc_signal` et `sc_fifo`. Notre classe `tlm_arbiter` hérite de la classe SystemC `sc_prim_channel` et implante la méthode virtuelle `update`. Cette méthode est appelée par le simulateur SystemC à la fin de chaque δ -cycle durant lequel la méthode `request_update` a été appelée.

Enfin, un tableau d'événements indexé par les identifiants, qui sont aussi utilisés comme priorités, permet de ne réveiller que le processus qui va pouvoir entrer dans la section critique. Nous réveillons toujours le processus dont l'identifiant est en tête de la liste `pending_requests`. Chaque fois qu'un processus est réveillé, nous retirons son identifiant de cette liste.

Le code complet est disponible à l'annexe A. Il s'agit de la version type "`tac_arbiter`"; la version "`tac_seq`" peut être obtenue en utilisant une *fifo* pour la liste `pending_requests`.

Pour utiliser cette classe avec l'exemple de la figure 4.8, il suffit de remplacer `sc_mutex` par `tlm_arbiter`, et d'ajouter le numéro de port lors de l'appel à la méthode `lock`. Cette nouvelle classe permet de déplacer l'appel à `unlock` dans la fonction `acknowledge`, et donc aussi de libérer le processus esclave plus tôt en supprimant l'événement `ack` et les instructions qui s'y rapportent.

4.4.2 Analyse de l'exemple et limitations

La première question était : *sait-on écrire des programmes indépendants à l'ordonnancement ?* Sur les deux exemples de programmes dépendants donnés dans ce chapitre, nous avons su les transformer en programme indépendant tout en conservant leur fonctionnalité. L'exemple de l'arbitrage montre cependant que cela nécessite un raisonnement relativement complexe, et du code supplémentaire à écrire. Le surcoût en temps de programmation n'est pas négligeable.

Ensuite, il s'agit de savoir comment prouver qu'un modèle est bien indépendant à l'ordonnancement. Une idée est de prouver que chaque composant est indépendant à l'ordonnancement, selon une définition et des hypothèses à préciser, puis à montrer que la composition reste indépendante. Pour un composant donné, nous ne disposons pas d'outils automatiques pour prouver l'indépendance, et la prouver manuellement n'est en général pas une trivialité (le lecteur peut essayer de prouver le contrôleur d'interruptions avec `tlm_arbiter` ci-dessus pour s'en convaincre). Pour prouver l'indépendance de la composition, la difficulté est de trouver les bonnes hypothèses et les bonnes garanties pour chaque composant. L'hypothèse sur la chronologie utilisée pour notre classe `tlm_arbiter` n'est, par exemple, pas assez forte pour un composant mémoire ; en effet, deux écritures à une même adresse et pendant un même δ -cycle doivent être ordonnées de façon déterministe.

Le dernier point, et finalement le plus important, est de savoir si écrire des programmes indépendants de l'ordonnancement est souhaitable. Rendre un programme indépendant revient à fixer un ordre aux événements observables de façon arbitraire. Or les programmes considérés sont des

modèles de système matériel avec logiciel embarqué. L'ordre de ces événements observables dans le système matériel est encore inconnu lors de l'écriture du modèle. Un ordre fixé trop tôt a donc toutes les chances de ne pas correspondre à la réalité. De plus, il se peut que l'indéterminisme de l'ordonnanceur corresponde à un véritable indéterminisme dans le système final, utilisé dans son environnement réel. Un modèle représentatif de la réalité doit donc être indéterministe. Ce troisième point serait aussi valable s'il s'était agi de créer une autre extension du langage C++ avec un ordonnanceur déterministe, comme cela a été fait pour JAVA [Bou02] ou ML [Man06].

4.4.3 Conclusion

Il s'avère que l'approche consistant à n'écrire que des modèles indépendants à l'ordonnancement n'est pas viable, pour des raisons de coût et surtout de réalisme. Nous devons donc admettre que le comportement observable d'un programme puisse varier en fonction des choix de l'ordonnanceur SystemC.

En dehors de quelques cas très particuliers, comme la réalisation de petit prototype avec une très courte durée de vie, il est souhaité qu'un modèle soit fonctionnellement correct pour tous les ordonnancements autorisés pour la spécification. Dans le cadre de la validation par le test de ces modèles, il est donc impératif de simuler chaque test, c'est-à-dire chaque jeu de données, avec plusieurs ordonnancements différents. Les chapitres suivants de cette thèse montrent comment couvrir l'espace des ordonnancements valides de façon efficace.

Chapitre 5

Génération automatique d'ordonnancements

Sommaire

5.1 Introduction	55
5.1.1 Objectif	55
5.1.2 Principe général	56
5.1.3 Contenu et plan du chapitre	56
5.2 Séparation de l'ordonnement et des données	57
5.2.1 Données fixées statiquement	57
5.2.2 Données générées dynamiquement	57
5.3 Représentation formelle	58
5.3.1 Système sous-test et ordonnancements	58
5.3.2 Relations entre les ordonnancements	59
5.3.3 Représentations graphiques	62
5.3.4 Égalité de transitions et lien avec le code source du SSTD	64
5.4 Algorithmes	66
5.4.1 Relation de commutativité	66
5.4.2 Ordre causal et permutabilité	70
5.4.3 Génération des nouveaux ordonnancements	71
5.5 Mise en application pour la validation	75
5.5.1 Propriété principale	75
5.5.2 Conséquences pour la validation	78

5.1 Introduction

5.1.1 Objectif

Nous avons vu au chapitre précédent qu'il pouvait exister pour un programme et des données fixées, plusieurs exécutions possibles, certaines correctes, d'autres non. Il serait en général impossible d'exécuter chaque test avec tous les ordonnancements légaux. L'objectif, dans ce chapitre, est de présenter une méthode qui ne génère que des ordonnancements pertinents.

5.1.2 Principe général

Le principe général est le suivant : nous commençons par exécuter le programme avec un ordonnancement quelconque. Chaque fois que nous suspectons que des choix d'ordonnancement mènent à des comportements différents, nous ré-exécutons le programme avec un nouvel ordonnancement. L'idée consiste à observer les actions effectuées par chaque processus afin de deviner si un ordre différent (et donc un ordonnancement différent) aurait pu mener à un résultat différent. Pour décider si un nouvel ordonnancement est nécessaire, nous utilisons un critère approximatif dans le sens suivant : nous pouvons engendrer plusieurs ordonnancements menant au même résultat, mais nous ne pouvons considérer comme équivalents deux ordonnancements qui mènent à des résultats différents. Le résultat final est un jeu d'ordonnements suffisamment riche pour offrir les garanties nécessaires pour la validation, et suffisamment concis pour être entièrement exécuté, y compris dans le cas de programmes réels.

Observer les actions de communications permet de déduire l'égalité de deux états atteints par des ordonnancements différents. Dans les modèles que nous considérons, comparer deux sauvegardes de l'état de la mémoire (des “*dump*”) avec ce même objectif, serait difficile techniquement et certainement trop coûteux.

Cette technique a été publiée dans [FG05], sous le nom de *réduction d'ordre partiel dynamique* (DPOR). Étant donné un état avec un ensemble E de transitions éligibles, les réductions d'ordre partiel *statique* permettent de *retirer* certaines de ces transitions de l'espace à explorer, en fonction de leurs communications *futures* [Pel93]. La réduction dynamique fonctionne, en quelque sorte, dans l'autre sens : un premier ordonnancement est simulé, puis une analyse des communications portant sur le *passé* de l'état courant permettent d'*ajouter* les ordonnancements pertinents. Les réductions d'ordre partiel statiques, basées sur les *persistent sets* et *stubborn sets*, nécessitent une analyse statique préalable pour le calcul des dépendances. Cette analyse oblige, dans le cas général, à des approximations. La réduction d'ordre partiel dynamique évite cette analyse statique préalable. En contrepartie, elle oblige à calculer effectivement l'ordre partiel qui définit la classe d'équivalence de l'exécution courante.

Par rapport à la technique décrite dans [FG05], nous avons dû apporter quelques adaptations nécessaires pour l'application à SystemC. Notamment, pour chaque structure spécifique à SystemC, il faut définir les cas de base pour le calcul des dépendances. De plus, nous avons modifié l'algorithme principal, qui repose maintenant sur une notion de *contraintes d'ordonnements*.

En 2006, d'autres travaux utilisant un principe similaire ont été publiés indépendamment [SA06, LC06]. Ils ont été disponibles trop tard pour que nous puissions nous en inspirer pour nos propres travaux.

5.1.3 Contenu et plan du chapitre

Ce chapitre présente notre méthode de génération automatique d'ordonnements d'un point de vue théorique. Le point le plus important étant de définir formellement ce que “*différent*” signifie. Les deux sections suivantes sont essentiellement indépendantes du langage utilisé, cependant la mise en œuvre (section 5.4) nécessite de considérer un langage particulier, dans notre cas SystemC. Les difficultés et autres détails techniques seront décrits au chapitre suivant ; l'efficacité pratique sera évaluée au chapitre 7.

La section 5.2 discute de la génération des ordonnancements indépendamment de celle des données. Les structures mathématiques nécessaires pour la description et la justification de notre méthode de génération sont définies dans la section 5.3. La section 5.4 fournit dans les grandes lignes

la description des algorithmes assurant la génération des ordonnancements. Enfin, l'utilisation de la méthode présentée pour la validation est justifiée formellement dans la section 5.5.

5.2 Séparation de l'ordonnancement et des données

La méthode présentée ne génère que des ordonnancements or en général un programme a aussi besoin de données pour fonctionner, comme par exemple un flux vidéo, ou des stimuli provenant de l'extérieur via un modèle d'UART¹. Il faut donc disposer de données ou d'un outil permettant de les générer dynamiquement. Nous discutons dans cette section de la recombinaison d'un test valide à partir de données et d'ordonnements générés séparément.

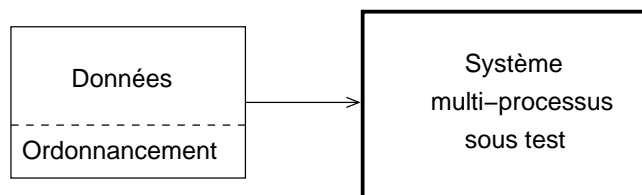


FIG. 5.1 – Un test = des données + un ordonnancement

5.2.1 Données fixées statiquement

Les données peuvent être générées statiquement. Il s'agit encore du cas le plus fréquent dans notre contexte. Les données peuvent soit se trouver dans un fichier et être lues à l'exécution, soit être intégrées aux sources C++ du modèle sous test.

Supposons par exemple que l'on souhaite tester un DMA². La méthode la plus courante consiste à l'intégrer à un modèle contenant un ISS³ et une mémoire, et à écrire du logiciel embarqué qui va le stimuler de la façon voulue.

Dans ce cas, quels que soient les ordonnancements générés, nous sommes certains de pouvoir les exécuter avec les mêmes données. Au pire, une exécution se bloquera avant la fin normale et une partie des données n'aura pas été consommée.

5.2.2 Données générées dynamiquement

Les données peuvent aussi être générées dynamiquement. Cela est nécessaire lorsque les entrées dépendent des sorties précédentes. Pour notre méthode de génération d'ordonnements, il est nécessaire que le générateur de données puisse fournir plusieurs fois les mêmes données, autrement dit les données doivent être reproductibles.

Un premier type de problème survient quand deux générateurs de données, intégrés via deux processus SystemC différents, accèdent à une même ressource, par un exemple un générateur pseudo-aléatoire. Dans ce cas, deux ordonnancements peuvent induire des données différentes, car les accès au générateur aléatoire auront lieu dans un ordre différent, y compris si son germe initial est le même. Il s'agit là d'une erreur de conception de l'environnement de test. Celui-ci doit alors être corrigé en instanciant un générateur pseudo-aléatoire avec germe indépendant pour chacun des générateurs de

¹UART = *Universal Asynchronous Receiver Transmitter*, gère les communications avec l'extérieur

²DMA = *Direct Memory Access*

³ISS = *Instruction Set Simulator*, modélise un processeur

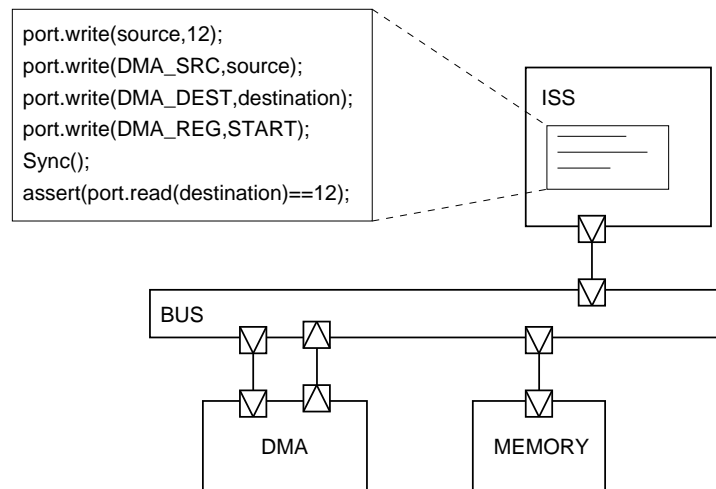


FIG. 5.2 – Plateforme pour le test d'un DMA

données. Plus généralement, des générateurs de données liés à des processus SystemC différents ne doivent pas dépendre des mêmes ressources, à moins que celles-ci fournissent les mêmes résultats quel que soit l'ordre des accès.

Un problème plus complexe peut aussi se produire quand les sorties et donc les entrées sont dépendantes de l'ordonnancement effectif. Il peut alors être impossible de réexécuter le système avec les mêmes données mais des ordonnancements distincts. En pratique, ce type de cas semble très rare lorsque l'on considère des programmes réels (nous en n'avons pas encore rencontrés). À noter que si une sortie est juste décalée dans le temps, les générateurs de données sont généralement robustes et décalent les entrées correspondantes en fonction, sans que cela nous pose de problème. Il n'est pas exclu que nous rencontrions dans l'avenir un cas réel qui oblige à une analyse plus poussée du problème.

5.3 Représentation formelle

Nous allons maintenant expliquer comment nous générons un jeu d'ordonnements pour des systèmes multitâches. Dans cette section et les suivantes, nous nous intéressons à un couple particulier composé du système sous-test et d'un jeu de données fixé, que l'on suppose valide pour tous les ordonnancements générés. On note *SST* les systèmes sous-test, et *SSTD* le système sous-test accompagné d'un jeu de données particulier. Ce qui suit ne dépend pas directement de l'utilisation ou non de SystemC.

5.3.1 Système sous-test et ordonnancements

Une exécution d'un *SSTD* est entièrement définie par son ordonnancement. Un *ordonnement* est une séquence d'identifiants de processus et de symboles δ et χ . Ces symboles δ et χ désignent respectivement un changement de δ -cycle ou un avancement du temps. On note P l'ensemble des identifiants de processus.

Un *état global* désigne le contenu de la mémoire utilisée par le *SSTD*, y compris la position dans le code de chaque processus.

Une vision très abstraite du SSTD va nous suffire : un SSTD est modélisé par une fonction des ordonnancements vers ses états globaux. Cette fonction est partielle puisque seuls certains éléments de $(P \cup \{\delta, \chi\})^*$ représentent des ordonnancements possibles pour le SSTD, cela à cause des contraintes de synchronisation entre les processus.

Définition 1 — Ordonnements

Soit M un SSTD. P_M désigne l'ensemble de ses processus ; S_M désigne l'ensemble de ses états globaux ; $F_M : (P_M \cup \{\delta, \chi\})^* \rightarrow S_M$ désigne la fonction partielle qui modélise M . Un ordonnancement est un élément de $(P_M \cup \{\delta, \chi\})^*$; un ordonnancement valide est un élément du domaine de définition de $F_M : A_M = D_{F_M} \subset (P_M \cup \{\delta, \chi\})^*$. Un ordonnancement est dit complet s'il ne peut être prolongé en un autre ordonnancement valide.

Par exemple, pour les programmes de la section 4.2, nous avons :

- $A_{\text{bozo}} = D_{F_{\text{bozo}}} = \{PQP\chi QP, PQP\chi PQ, QP\chi Q\}$
- $F_{\text{bozo}++}(PQR) = F_{\text{bozo}++}(PRQ) = F_{\text{bozo}++}(RPQ)$

Définition 2 — Transitions

Une transition est une exécution d'un processus pour un ordonnancement particulier. Chaque transition d'un ordonnancement est identifiée par l'identifiant du processus, que l'on indice par le numéro d'occurrence de ce processus au sein de l'ordonnancement considéré.

Une transition correspond à une section de code atomique pour l'ordonnancement SystemC. Autrement dit, il s'agit de l'ensemble du code exécuté entre le moment où l'ordonnancement élit un processus, et le moment où le processus lui rend la main (via une instruction `wait` ou `yield`). Une transition peut donc couvrir une longue section de code, réaliser de multiples accès à la mémoire et appeler des fonctions. Il est possible qu'une transition se termine à l'intérieur de l'appel d'une fonction, éventuellement dans un module SystemC distinct.

Par exemple, l'ordonnancement pqp contient trois transitions que l'on note, dans l'ordre : p_1 , q_1 et p_2 .

Définition 3 — Permutations

Soit $u = vp_iwq_j$ un ordonnancement valide, dans lequel p_i (respectivement q_j) désigne la i -ème (respectivement j -ème) exécution du processus p (respectivement q), tel que défini ci-dessus. Permuter les transitions p_i et q_j signifie générer un nouvel ordonnancement valide u' commençant par v et tel que la j -ème transition de q soit avant la i -ème transition de p : $\exists x, y, u' = vxq_jyp_i$. L'ordonnancement u' est appelé une permutation de p_i et q_j pour u .

Il est important de noter que, avec les notations de la définition ci-dessus, u' n'a pas nécessairement la même longueur que u . Certaines transitions de w peuvent être absentes à la fois de x et y , soit par choix, soit parce que les processus correspondant ne sont pas éligibles à l'issue de l'ordonnancement u' .

Dans la suite on utilisera les lettres p, q, r, \dots pour désigner des processus, les lettres a, b, c, \dots pour désigner les transitions et enfin les lettres u, v, w, \dots pour désigner les ordonnancements ou portions d'ordonnements. Les indices seront omis quand ils pourront être trivialement déduits du contexte.

5.3.2 Relations entre les ordonnancements

La plupart des définitions de cette sous-section sont standard dans la littérature sur les réductions d'ordre partiel.

5.3.2.1 Relation d'équivalence

Nous allons tout d'abord définir une relation d'équivalence sur les ordonnancements. L'objectif sera ensuite de générer au moins un représentant de chaque classe d'équivalence. Intuitivement, si deux ordonnancements mènent au même état final, alors il n'est pas utile de les exécuter tous les deux. Pour nos objectifs de validation, il suffirait de prendre comme équivalence “ u et v sont équivalents si et seulement si $F_M(u) = F_M(v)$ ”. Cependant, afin de pouvoir générer et garantir que l'on a au moins un représentant par classe, nous allons avoir besoin d'une relation d'équivalence plus fine.

Il nous faut tout d'abord définir la relation “ \sim ” :

Définition 4 — Équivalence locale

|| Quels que soient $uabv \in A_M$, $uabv \sim ubav$ si et seulement si ($ubav \in A_M \wedge F_M(uabv) = F_M(ubav)$).

La relation d'équivalence dont nous avons besoin en découle directement :

Définition 5 — Équivalence d'ordonnements

|| La relation d'équivalence sur les ordonnancements, noté “ \equiv ”, est la fermeture réflexive et transitive de la relation “ \sim ”.

Cette définition garantit la propriété :

$$\forall u, v \in A_M, u \equiv v \Rightarrow F(u) = F(v)$$

Par conséquent, si on génère un élément de chaque classe d'équivalence, nous connaîtrons tous les états finals accessibles. Nous verrons à la section 5.5 comment cela peut être utilisé pour la validation d'un SSTD.

La classe d'équivalence d'un ordonnancement u est noté $[u]$.

5.3.2.2 Relations nécessaires pour la génération

Nous devons générer un nouvel ordonnancement seulement quand la permutation de deux transitions peut résulter en un ordonnancement qui ne sera pas équivalent à ceux déjà générés.

Supposons que l'on soit en train d'exécuter un SSTD et que les deux derniers processus exécutés étaient p suivi de q . Cela peut se noter formellement : $u = u_1 p_i q_j$. Si aucune raison causale ne l'empêche, par exemple si le processus q n'était pas en attente d'un événement notifié par p , alors il est possible de permuter ces deux dernières transitions. Dans ce cas, exécuter q à la place de p quand on se trouve dans l'état $F_M(u_1)$, peut éventuellement mener à un chemin d'exécution divergent, comme illustré par la figure 5.3. La question qui nous préoccupe alors est : “Est-ce que ces deux ordonnancements peuvent mener à des états finals différents?”, ou plus formellement : “ $F_M(u_1 p q) = F_M(u_1 q p)$?”. Il faut bien noter que nous devons répondre à cette question *sans* exécuter entièrement $u_1 q p$. Par conséquent nous nous basons sur les actions de communication effectuées par chacun des deux processus pour extrapoler la réponse à cette question. Nous ne pouvons pas répondre avec certitude dans le cas général. La règle est que nous générons un nouvel ordonnancement chaque fois que nous ne pouvons pas prouver que la permutation est sans effet sur la suite de l'exécution.

Il nous faut étudier deux sous-questions :

- quelles transitions *pouvons* nous permuter ?
- quelles transitions est-il *utile* de permuter ?

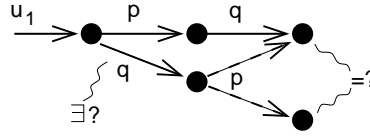


FIG. 5.3 – Un éventuel chemin divergent. Les cercles noirs pleins représentent des états globaux du SSTD.

Pour répondre à la première question, nous allons définir une relation appelée *permutabilité*, et pour la deuxième nous allons définir une relation d'*indépendance*.

La théorie liée aux *réductions d'ordre partiel* repose sur la définition de *transitions dépendantes* [Maz87]. Étant donné un ordonnancement valide u , deux transitions a et b sont *indépendantes* si aucune des deux n'a été activées par l'autre, et si leur permutation donne un ordonnancement valide qui mène toujours au même état final. On rappelle qu'en SystemC, un processus actif, c'est-à-dire éligible, ne peut être désactivé par un autre processus ; autrement dit il ne peut redevenir non-éligible sans avoir été exécuté. On nomme \mathcal{D} l'ensemble des paires de transitions dépendantes, et \mathcal{I} l'ensemble complémentaire des transitions indépendantes.

Définition 6 — Dépendance des transitions

Soient M un SSTD $u_1 a u_2 b u_3 \in A_M$ un ordonnancement valide, $(a, b) \in \mathcal{I}$ si et seulement si :

$$\forall u_1 v_1 a b v_2 \equiv u_1 a u_2 b u_3, u_1 v_1 b a \in A_M \text{ et } F_M(u_1 v_1 b a) = F_M(u_1 v_1 a b)$$

Étant donné un ordonnancement u contenant deux transitions p_i et q_j , nous notons $p_i <_u q_j$ si la transition p_i (c'est-à-dire la i -ème exécution du processus p) a eu lieu avant la transition q_j (j -ème exécution du processus q). La relation $<_u$ décrit l'ordre complet des transitions d'un ordonnancement particulier. L'*ordre causal* décrit quant à lui l'ordre des transitions pour toute une classe d'équivalence. C'est un ordre partiel. Il aussi appelé "*happens-before relation*" dans la littérature.

Définition 7 — Ordre causal

Les transitions a et b d'un ordonnancement valide u tel que $a <_u b$ sont *causalement ordonnées* si et seulement si :

$$\forall v \equiv u, a <_v b$$

Dans ce cas, on note : $a < b$.

La relation d'ordre causal ne va pas nous servir directement dans la suite, mais elle nous sera utile pour calculer la relation de *permutabilité*.

Deux transitions sont dites *permutables* s'il est possible de les permuter sans permuter d'autres paires de transitions dépendantes. Contrairement à la relation d'ordre causal, la relation de permutabilité n'est pas une relation d'ordre. nous nommons \mathcal{P} l'ensemble des paires de transitions permutable.

Définition 8 — Permutabilité

Une paire de transitions (a, b) d'un ordonnancement valide $u = u_1 a u_2 b u_3$ est *permutable* si et seulement si :

$$\exists v_1, v_2 \text{ tel que } u_1 v_1 a b v_2 \equiv u \text{ et } u_1 v_1 b \in A_M$$

Autrement dit deux transitions sont permutable si et seulement si :

1. Il existe un ordonnancement équivalent dans lequel elles sont consécutives.
2. Le processus de la seconde transition peut être élu à la place du processus de la première transition dans cet ordonnancement équivalent.

Dans la suite, nous aurons besoin de la notion de *contrainte d'ordonnement* pour décrire la "frontière" des classes d'équivalence, puis pour maintenir un historique de l'espace des ordonnancements déjà couvert.

Définition 9 — contrainte d'ordonnement

Une contrainte d'ordonnement est un quadruplet (p, i, q, j) que l'on note sous la forme " $p_i < q_j$ ". Une contrainte d'ordonnement " $p_i < q_j$ " est vérifiée par un ordonnancement u si et seulement si : $q_j \in u \Rightarrow p_i \in u \wedge p_i <_u q_j$. Dans ce cas, on note $u \models "p_i < q_j"$.

Autrement dit, la contrainte " $p_i < q_j$ " est vérifiée par un ordonnancement, soit si le processus q a été exécuté moins de j fois, soit si sa j -ème exécution a eu lieu après la i -ème exécution du processus p . Un ordonnancement vérifie un ensemble de contraintes d'ordonnement s'il vérifie chacune d'entre elles.

Nous aurons aussi besoin d'estimer la permutabilité de deux transitions en présence d'un ensemble C de contraintes d'ordonnement à respecter.

Définition 10 — Permutabilité sous contraintes

Une paire de transitions (p_i, q_j) d'un ordonnancement valide $u = u_1 p_i u_2 q_j u_3$ est permutable sachant C si et seulement si :

$$\begin{aligned} \exists v_1, v_2 \quad & \text{tel que} \quad u_1 v_1 p_i q_j v_2 \equiv u \\ & \text{et} \quad u_1 v_1 p_i q_j v_2 \models C \\ & \text{et} \quad "q_j < p_i" \notin C \\ & \text{et} \quad u_1 v_1 q_j \in A_M \end{aligned}$$

Dans ce cas, on note $(p_i, q_j) \in \mathcal{P}|C$.

Pour résumer, étant donné une paire de transitions, il y a trois possibilités différentes :

1. soit elles sont indépendantes ;
2. soit elles sont dépendantes et non-permutables ;
3. soit elles sont dépendantes et permutables.

Dans le premier cas, il est *inutile* de les permuter ; dans le deuxième cas, il est *impossible* de les permuter. Par contre, dans le troisième et dernier cas, la permutation est possible et peut conduire à un état final différent. Notre objectif à la section suivante sera donc d'arriver à détecter l'ensemble, ou un sur-ensemble, des paires de transitions dépendantes et permutables.

5.3.3 Représentations graphiques

Nous représentons les communications entre les processus par deux types de graphique, l'un statique, l'autre dynamique. Ces graphiques facilitent la compréhension d'un système qu'un développeur ou testeur découvre. Le graphique des dépendances statiques permet de voir en un coup d'oeil qui communique avec qui. Le graphique des dépendances dynamiques aide à comprendre ce qui s'est réellement passé pour une exécution donnée.

5.3.3.1 Graphe des dépendances statiques

Un *Graphe des Dépendances Statiques* (GDS) représente les communications possibles entre deux processus d'un système. Il est construit à partir du code de chaque processus et ne dépend pas d'une exécution particulière.

A chaque processus correspond un noeud du graphe. Les arcs du graphe représentent les communications possibles entre deux processus : chaque fois qu'une affectation (ou assimilée) d'une variable partagée apparaît dans un processus p et qu'une lecture de cette même variable est présente dans un processus q , nous ajoutons au graphe un arc allant de celui qui écrit (ici p) à celui qui la lit (q). On procède de la même façon pour les événements en ajoutant un arc allant du processus qui notifie l'événement à celui qui l'attend.

La figure 5.4 donne le graphe des dépendances statiques pour le programme `bozo++` (cf figure 4.5). Il permet notamment de voir que le processus R est indépendant des deux autres.

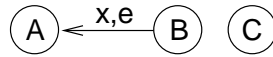


FIG. 5.4 – Graphe des dépendances statiques pour le programme `bozo++`

5.3.3.2 Graphe des dépendances dynamiques

Un *Graphe des Dépendances Dynamiques* (GDD) représente les communications effectives entre deux transitions d'une exécution. Il est construit à partir de l'observation des actions de communication. Contrairement au graphe des dépendances statiques, il y a autant de graphes des dépendances dynamiques que d'ordonnements valides.

A chaque ligne horizontale correspond un processus. Les nouveaux δ -cycles ou changement de cycle temporel sont représentés respectivement par une ou deux barres verticales. Les noeuds rectangulaires représentent les transitions des processus. Les flèches entre deux noeuds rectangulaires indiquent que les transitions correspondantes sont causalement ordonnées. Si les transitions sont permutable, la flèche est en pointillé (ou en rouge quand on dispose de la couleur), sinon le trait est plein. Par souci de lisibilité, on omet généralement les flèches quand elles sont obtenues par transitivité.

La figure 5.5 fourni le graphe des dépendances dynamiques pour l'exécution du programme `bozo` avec l'ordonnement $PQP \parallel QP$ (cf figure 4.5 et tableau 4.1).

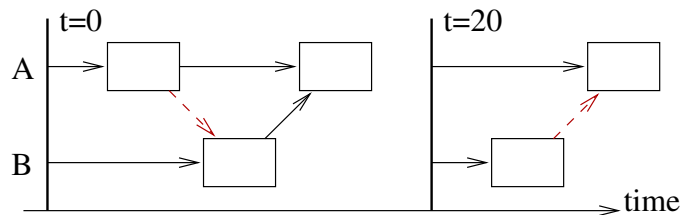


FIG. 5.5 – Graphe des dépendances dynamiques pour une exécution de `bozo`

Intuitivement, jouer sur l'ordonnement revient à déplacer horizontalement les noeuds rectangulaires. Les flèches limitent le déplacement des noeuds. En effet, leur projection sur l'axe horizontal doit toujours conserver le même sens pour que l'ordonnement correspondant reste dans la même classe d'équivalence. Il est possible de permuter des noeuds reliés par une flèche pointillée rouge mais, dès lors, nous changeons de classe d'équivalence et il se peut que la partie du graphe situé à droite ne

corresponde plus à un ordonnancement valide. Notre méthode de génération est fortement liée à cela : nous générons un nouvel ordonnancement pour chaque trait rouge qui n'a pas déjà été permuté.

5.3.4 Égalité de transitions et lien avec le code source du SSTD

Lors d'une transition, un processus du SSTD exécute une section de code et accède à des objets internes et partagés. Considérons deux ordonnancements u et v contenant chacun une transition p_i . Bien qu'étant dans les deux cas la i -ème transition du processus p , il se peut qu'elle corresponde à l'exécution d'un code différent, ou du même code mais avec des valeurs lues différentes, selon que l'on considère u ou v . Intuitivement, le comportement d'une transition dépend de ce qui s'est passé avant. Dans la suite, on notera $p_{i,u}$ la transition p_i de u , et $p_{i,v}$ la transition p_i de v . L'objectif de cette section est de donner des conditions suffisantes pour affirmer que les transitions $p_{i,u}$ et $p_{i,v}$ sont égales, c'est-à-dire qu'elles correspondent à l'exécution d'une même section de code, et qu'elles accèdent aux mêmes valeurs.

Cette sous-section n'est pas primordiale pour la suite du document, mais clarifie la notion de transition.

On définit la notion d'*égalité de transitions* à partir des deux cas élémentaires ci-dessous.

Définition 11 — Égalité de transitions

Étant donné les deux relations temporaires $=_\alpha$ et $=_\beta$ suivantes :

- $\forall p_{i,u}, p_{i,v}, p_{i,u} =_\alpha p_{i,v} \Leftrightarrow \exists w \text{ tel que } v = uw \vee u = vw.$
- $\forall p_{i,u}, p_{i,v}, p_{i,u} =_\beta p_{i,v} \Leftrightarrow u \equiv v$

Deux transitions a et b sont égales, noté $a = b$, si et seulement si le couple (a, b) appartient à la fermeture transitive de $\{x, y | x =_\alpha y \vee x =_\beta y\}$.

La première règle formalise juste que le comportement d'un programme ne dépend pas de son futur. La deuxième règle précise que les transitions de deux ordonnancements équivalents sont égales deux à deux ; il est clair que si une transition p_i pouvait correspondre à un code différent dans u et v avec $u \equiv v$, alors on n'aurait aucune garantie d'obtenir le même état final, idem pour les valeurs accédées par le code exécuté. En conséquence, si $p_{i,u} = p_{i,v}$ et $q_{j,u} = q_{j,v}$, alors $(p_{i,u}, q_{j,u}) \in \mathcal{C} \Leftrightarrow (p_{i,v}, q_{j,v}) \in \mathcal{C}$.

Deux choses sont à noter à propos de cette définition :

- L'union des deux relations $=_\alpha$ et $=_\beta$ est déjà réflexive et symétrique. C'est donc aussi le cas de sa fermeture transitive.
- Deux transitions p_i et p_j ne seront jamais considérées comme égales si $i \neq j$. Pouvoir les considérer comme égales dans certains cas nécessiterait de tenir compte des boucles dans le code source du processus.

La propriété suivante découle directement de la définition :

Propriété 1 Soient $u = u_1 p_i u_2$ et $v = v_1 p_i v_2$ deux ordonnancements. s'il existe des sections d'ordonnements u', v', w tels que $u \equiv w p_i u'$ et $v \equiv w p_i v'$, alors $p_{i,u} = p_{i,v}$.

Preuve : $p_{i,u} =_\beta p_{i,(w p_i u')} =_\alpha p_{i,(w p)} =_\alpha p_{i,(w p_i v')} =_\beta p_{i,v}$ et donc par transitivité $p_{i,u} = p_{i,v}$. ■

La définition ci-dessus n'est guère pratique pour évaluer, en pratique, l'égalité de transitions p_i d'ordonnements différents. La notion de *passé d'une transition* va nous permettre d'améliorer cela.

Définition 12 — Passé d'une transition

Le passé d'une transition p_i d'un ordonnancement u est l'ensemble $\Pi(p_{i,u}) = \{q_j \in u \mid q_j \prec p_i\}$.

La suite de cette section a pour objectif de montrer que deux transitions sont égales (selon la définition ci-dessus) si et seulement si elles ont le même passé. Une première conséquence intéressante est que si les transitions de deux ordonnancements ont le même passé deux à deux, alors ces ordonnancements sont équivalents.

Propriété 2 Pour tous ordonnancements u et v tel que :

$$\forall p_i \in u, p_i \in v \wedge \Pi(p_{i,u}) = \Pi(p_{i,v})$$

et réciproquement :

$$\forall p_i \in v, p_i \in u \wedge \Pi(p_{i,u}) = \Pi(p_{i,v})$$

alors les ordonnancements u et v sont équivalents : $u \equiv v$.

Preuve : On écrit u et v sous la forme $u = wp_iu'$ et $v = wv'p_iv''$ ou w est le préfixe commun, éventuellement vide, de u et v . On sait que : $\Pi(p_{i,v}) = \Pi(p_{i,u}) \subset w$ et donc $p_{i,v}$ est indépendante de chaque transition de v' . Par conséquent : $v \equiv v_1 = wp_iv''$. Le nouveau préfixe commun w_1 de u et v_1 est plus long que w d'au moins un élément. En appliquant récursivement la même transformation sur v_1 , on obtient une séquence $v \equiv v_1 \equiv \dots \equiv v_n$ tel que $v_n = u$ et donc $v \equiv u$. La longueur n de cette séquence est bornée par la longueur de v et u . ■

Nous allons aussi avoir besoin de la propriété suivante :

Propriété 3 Pour tout ordonnancement u et transition a de u , il existe v, w tels que $u \equiv vaw$ et $x \in v \Leftrightarrow x \in \Pi(a)$.

Informellement, cette propriété dit que toutes les transitions n'appartenant pas au passé d'une transition peuvent être permutées jusque derrière cette transition sans changer de classe d'équivalence.

Preuve : On note $P = \Pi(a)$ le passé de a dans u . L'ordonnancement u peut s'écrire sous la forme $u = u'au''$. Si $u' \subset P$, alors la propriété est trivialement vérifiée, sinon : soit x le dernier élément de $u' \setminus P$, l'ordonnancement u s'écrit alors sous la forme $u = wxvau''$ avec $v = b_1 \dots b_n$, $v \subset p$ et $x \notin P$. Par transitivité de l'ordre causal, on obtient aussi que $\forall i, (x, b_i) \in \mathcal{J}$. Par conséquent :

$$u = wxb_1 \dots b_n au'' \equiv wb_1xb_2 \dots b_n au'' \equiv wb_1 \dots b_n x au'' \equiv wb_1 \dots b_n axu'' = wvaxu''$$

Ainsi, on a obtenu un ordonnancement $u_1 = u'_1 au''_1 \equiv u$ tel que u'_1 soit plus court d'un élément que u' . Soit $m = |u'| - |P|$ le nombre de transitions de $u' \setminus P$, en appliquant m fois la séquence de permutations décrite ci-dessus, on obtient une séquence d'ordonnancements équivalents $u \equiv u_1 \equiv \dots \equiv u_m$ tel que $u_m = u'_m au''_m$ et $x \in u'_m \Leftrightarrow x \in P$. ■

Pour finir, voici enfin la propriété liant l'égalité des passés à l'égalité des transitions.

Propriété 4 Soient u et v deux ordonnancements, on a :

$$\begin{aligned} \forall p_{i,u}, p_{i,v}, p_{i,u} = p_{i,v} &\Leftrightarrow \forall q_j \in \Pi(p_{i,u}), q_j \in \Pi(p_{i,v}) \wedge \Pi(q_{j,u}) = \Pi(q_{j,v}) \\ &\wedge \forall q_j \in \Pi(p_{i,v}), q_j \in \Pi(p_{i,u}) \end{aligned}$$

Preuve :

Sens \Rightarrow

D'après la propriété 3 :

- $\exists u', u'', u \equiv u'p_i u''$ et $\forall a, a \in u' \Leftrightarrow a \in \Pi(p_{i,u})$
- $\exists v', v'', v \equiv v'p_i v''$ et $\forall b, b \in v' \Leftrightarrow b \in \Pi(p_{i,v})$

Or $\Pi(p_{i,u}) = \Pi(p_{i,v})$, donc d'après la propriété 2 $u' \equiv v'$, et donc $u \equiv v'p_i u''$ puis $p_{i,u} = p_{i,v}$ par la propriété 1.

Sens \Leftarrow :

La relation $=_\alpha$ conserve le passé : $\forall p_{i,u}, p_{i,v}, p_{i,u} =_\alpha p_{i,v} \Rightarrow \Pi(p_{i,u}) = \Pi(p_{i,v})$.

De même pour $=_\beta$ puisque l'ordre causal est constant au sein d'une classe d'équivalence, et donc par transitivité deux transitions égales ont le même passé. ■

Cette dernière propriété peut aussi s'exprimer sous cette forme : $\forall p_{i,u}, p_{i,v}, p_{i,u} = p_{i,v} \Leftrightarrow \Pi(p_{i,u}) \cong \Pi(p_{i,v})$, où \cong est l'extension ensembliste de l'égalité des transitions.

5.4 Algorithmes

Dans cette section nous ne donnons que les principes des algorithmes nécessaires à la construction des relations précédemment définies. Leur mise en œuvre sera détaillée au chapitre 6. Pour chaque paire de transitions, nous regardons d'une part si une éventuelle permutation peut mener à un état final différent, et d'autre si cette permutation est possible.

5.4.1 Relation de commutativité

Nous calculons ici une relation binaire de *commutativité* \mathcal{C} telle que si une paire de transition (a, b) est à la fois commutative et permutable, alors les transitions a et b sont aussi indépendantes. Ainsi, pour détecter les transitions dépendantes et permutable, il ne sera utile d'évaluer la permutableté de deux transitions, que si elles n'ont pas déjà été évaluées comme commutatives. Concrètement, nous supposons que toutes les paires de transitions sont permutable et évaluons si leur permutation mène ou non à un état global différent. La relation de commutativité a l'avantage de pouvoir se calculer en ne considérant que 2 transitions à la fois.

La question de la commutativité des transitions de type "changement de cycle" (δ ou χ) ne se pose pas puisque, comme nous le verrons dans la suite, ces transitions ne sont permutable avec aucune autre.

La commutativité de deux transitions dépend des actions de communications qui ont été exécutées par chacun des deux processus pendant ces transitions. Deux transitions ne sont pas commutatives si elles contiennent une paire d'actions qui ne sont pas commutatives, comme par exemple une lecture et une écriture sur une même variable. Intuitivement, deux actions sont commutatives si elles modifient l'état global de la même façon quel que soit le contexte et l'ordre dans lequel elles sont exécutées. Nous notons Γ la relation de *commutativité des actions*.

L'ordre des actions au sein d'un transition n'a pas d'importance. De même, le fait qu'une action ait été effectuée plusieurs fois n'a aucun effet. Autrement dit, nous nous intéressons seulement à l'ensemble des actions de communications ayant été exécuté au moins une fois au cours de la transition.

Définition 13 — Calcul de la relation de commutativité \mathcal{C}

Soient a et b deux transitions d'un ordonnancement $uavbw$, distinctes de δ et χ ; soient $\{\alpha_1, \dots, \alpha_m\}$ les actions de communications présentes dans la transition a , et $\{\beta_1, \dots, \beta_n\}$ celles présentes dans b ; alors $(a, b) \in \mathcal{C}$ si et seulement si $\forall (i, j) \in [1..m] \times [1..n], (\alpha_i, \beta_j) \in \Gamma$.

Il ne nous reste maintenant plus qu'à définir la relation Γ pour les deux types d'objets qui nous intéressent : les variables partagées et les événements.

5.4.1.1 Variables

La commutativité pour les variables n'est pas spécifique à SystemC. L'analyse se faisant pour une exécution particulière, nous possédons toutes les informations souhaitables. D'une part nous savons si la valeur a été modifiée lors d'une écriture. D'autre part, dans le cas d'un accès à un tableau, nous connaissons toujours la valeur de l'index. Plus généralement, si le SSTD utilise des pointeurs, chaque case mémoire accessible par au moins deux processus peut être considérée une variable. Comme la case mémoire accédée est toujours identifiée, aucune abstraction n'est nécessaire.

Nous considérons deux types d'actions pour les variables, plus une variante. Le premier type d'action est la *lecture*, que l'on symbolisera par la lettre R comme "*Read*". Par lecture, nous désignons toutes les opérations qui permettent de connaître la valeur d'une variable. Le deuxième type d'action est l'*écriture*, que l'on symbolisera par la lettre W comme "*Write*". Par écriture, nous désignons toutes les opérations qui permettent de modifier la valeur d'une variable, l'exemple le plus courant étant l'affectation. Enfin, nous ajoutons un troisième type : l'*écriture non-modifiante*, symbolisée par la lettre T comme "*Touch*". Une écriture non-modifiante consiste à affecter une variable partagée avec la valeur qu'elle possédait déjà. Considérer les écritures non-modifiantes comme des écritures modifiantes n'aurait comme conséquence que de générer des ordonnancements redondants. Par contre, ignorer complètement les écritures non-modifiantes serait une erreur car celles-ci peuvent devenir des écritures modifiantes avec un ordonnancement différent.

Le tableau 5.1 fournit la valeur de la relation Γ pour chaque couple d'actions qui s'appliquent à la même variable. Toutes les paires d'actions qui concernent des variables différentes sont commutatives. L'ordre des deux actions est importantes : $(M_v, T_v) \in \Gamma$ mais $(T_v, M_v) \notin \Gamma$. Dans le premier cas, il s'agit de deux affectations de la même valeur. Dans le deuxième cas, il s'agit d'affectations de valeurs différentes dont la première est égale à l'ancienne valeur de la variable.

1ère \ 2ème	R	M	T
R	Γ	\mathcal{F}	Γ
M	\mathcal{F}	\mathcal{F}	Γ
T	Γ	\mathcal{F}	Γ

TAB. 5.1 – Commutativité des opérations portant sur une même variable

Les deux premiers exemples correspondent aux cas non-commutatifs (R, M) et (M, R) , puis (T, M) et (M, M) . Les deux exemples suivants correspondent aux cas commutatifs (R, T) et (T, R) , puis (M, T) . Dans tous les exemples, g désigne une variable partagée et n est une variable locale.

EXEMPLE 1 — Lecture - écriture

Initialement : $g = 42$

Transition a : $n = g$;

Transition b : $g = 12$;

Avec l'ordre ab , la variable n vaut finalement 12 (cas (R, M)).
 Avec l'ordre ba , la variable n vaut finalement 42 (cas (M, R)).

—

EXEMPLE 2 — Deux écritures

Initialement : $g=2$

Transition a : $g = 2$;

Transition b : $g = 4$;

Avec l'ordre ab , la variable g vaut finalement 4 (cas (T, M)).

Avec l'ordre ba , la variable g vaut finalement 2 (cas (M, M)).

—

EXEMPLE 3 — Lecture - écriture constante

Initialement : $g=12$

Transition a : $n = g$;

Transition b : $g = 12$;

Quel que soit l'ordre, la variable n vaut finalement 12 (cas (R, T) et (T, R)).

—

EXEMPLE 4 — Écritures identiques

Initialement : $g=42$

Transition a : $g = 12$;

Transition b : $g = 12$;

Quel que soit l'ordre, la variable g vaut finalement 12 et seule la première écriture est modifiante (cas (M, T)).

—

5.4.1.2 Événements SystemC

Il y a deux opérations sur les événements SystemC : l'*attente* (W) et la *notification* (N). Ces deux opérations correspondent respectivement aux instructions `wait` et `notify` de SystemC. Seules les notifications immédiates nous intéressent ; les notifications avec délai ont le même effet quel que soit l'ordonnement.

Le tableau 5.2 fournit la valeur de la relation Γ pour chaque couple d'actions qui s'appliquent au même événement. Comme pour les variables partagées, seules les paires d'actions qui concernent le même événement, peuvent ne pas être commutatives. Il faut noter que cela n'est plus vrai pour les notifications dès que l'on est en présence d'attentes sur liste d'événements (par exemple `wait (elf)`).

	W	N
W	Γ	\mathcal{F}
N	\mathcal{F}	\mathcal{F}

TAB. 5.2 – Commutativité des opérations portant sur un même événement

L'absence de commutativité entre une attente et une notification est claire : si l'attente vient d'abord, alors le processus correspondant va être réveillé par la notification ; dans le cas contraire ce processus va "rater" la notification et rester en attente jusqu'à l'éventuelle prochaine notification. L'exemple ci-dessous illustre ce cas.

EXEMPLE 5 — Attente - Notification

Transition p_i : `wait (e) ;`Transition q_j : `e.notify () ;`Avec l'ordre $p_i q_j$, le processus p est finalement éligible (cas (W, N)).Avec l'ordre $q_j p_i$, le processus p est finalement en attente sur e (cas (N, W)).

—

La dépendance entre deux notifications est plus difficile à comprendre car elle est due, non pas à l'état global du système, mais au calcul de la relation d'ordre causal. L'exemple ci-dessous illustre le problème.

EXEMPLE 6 — Notifications non-commutatives

Initialement, le processus p est en attente sur l'événement e et les processus q et r sont éligibles.

- Code du processus p : `cout << 'a' ; x = 1 ;`
- Code du processus q : `cout << 'b' ; x = 2 ; e.notify () ;`
- Code du processus r : `cout << 'c' ; e.notify () ;`

Il y n'a qu'une transition pour chaque processus. On nomme les trois transitions a , b et c selon la lettre qu'elles affichent lors de leur exécution. Il y a quatre ordonnancements valides : bac , bca , cba et cab . Quand la transition b est exécutée avant c , le couple de transitions (a, b) n'est pas dans C mais a et b sont causalement ordonnées car le processus p de a a été réveillé par le processus q de b . Cependant, si l'on permute les transitions b et c , alors b n'est plus causalement avant a puisque p est dans ce cas réveillé par c et non plus par b . En conséquence, la paire de transition (b, a) devient non-commutative au sens strict.

—

En résumant, permuter deux notifications ne modifie pas l'état global du système lui-même, mais modifie une partie de l'historique, qui est enregistrée et utilisée par l'analyseur pour générer les nouveaux ordonnancements. Il est possible d'être plus précis (c'est-à-dire d'éviter de générer des ordonnancements redondants) en distinguant les notifications qui ont réveillé un processus, de celles qui n'ont réveillé personne. Nous utilisons finalement la règle suivante, qui est adaptée aux attentes sur liste d'événements : si une notification d'un événement e réveille un processus p qui attendait à ce moment une liste d'événements L , alors cette notification est non-commutative avec toute notification d'un événement f appartenant à L . Nous ne traitons pas les instructions d'attentes de la forme `wait (e&f)`, c'est-à-dire avec liste conjonctive.

EXEMPLE 7 — Attente sur liste d'événements

Il s'agit, à peu de chose près, du même programme que ci-dessus : les comportements sont semblables mais nous utilisons cette fois deux événements distincts.

- Code du processus p : `wait (e|f) ; x = 1 ;`
- Code du processus q : `wait (12) ; x = 2 ; e.notify () ;`
- Code du processus r : `wait (12) ; f.notify () ;`

Considérons l'ordonnement $p_1 q_1 r_1 r_2 q_2 p_2$. Dans ce cas, après p_1 , le processus p attend un événement quelconque parmi $L = \{e, f\}$. Il est réveillé par une notification présente dans la transition r_2 . D'après la règle ci-dessus, cette notification de f est donc non-commutative avec les notifications de f et e , y compris celle présente dans la transition q_2 . En conséquences, les transitions r_2 et q_2 sont dépendantes.

—

5.4.2 Ordre causal et permutabilité

Le calcul de la relation de permutabilité est basé sur celui de l'ordre causal. Le calcul de l'ordre causal se fait pas par pas. Nous notons $prec(u)$ l'ensemble de paires de transitions correspondant à l'ordre causal pour u : $(a, b) \in prec(u) \Leftrightarrow a \prec_u b$.

Initialement, la relation d'ordre est vide : $prec(\varepsilon) = \emptyset$.

Avant de donner l'itération du calcul, nous devons encore définir une relation supplémentaire. Soient a et b deux transitions, nous savons que a et b ne sont pas permutable, et qu'en conséquence $a \prec b$ au moins dans les *trois cas élémentaires* ci-dessous :

- les transitions a et b appartiennent au même processus (les permuter n'aurait alors pas de sens)
- a ou b correspondent à un changement de cycle (δ ou χ)
- le processus de la transition b a été réveillé par la transition a .

Si l'objectif est de calculer $\mathcal{P}|C$ (défini à la section 5.3.2.2), où C est un ensemble de contraintes d'ordonnement à respecter, il faut ajouter un quatrième cas :

- a et b sont ordonnées par une contrainte de C (" $p_i < q_j$ " $\in C$ avec $a = p_i$ et $b = q_j$).

Quand nous sommes dans l'un de ces cas, nous notons : $(a, b) \in NP$.

Ce qui suit est une adaptation de [FG05].

Connaissant $prec(u)$, nous calculons $prec(ub)$ de la façon suivante :

$$prec_1(ub) = prec(u) \cup \{(a, b) \in u \times \{b\} | (a, b) \in NP\} \quad (5.1)$$

$$prec_2(ub) = prec_1(ub) \cup \{(a, b) \in u \times \{b\} | (a, b) \notin \mathcal{C}\} \quad (5.2)$$

$$prec(ub) = \text{fermeture transitive de } prec_2(ub) \quad (5.3)$$

Finalement, nous avons $(a, b) \in \mathcal{P}$ dans $u_1 a u_2 b u_3$ si et seulement si (a, b) n'appartient pas à la fermeture transitive de $prec_1(u_1 a u_2 b)$.

EXEMPLE 8 — Programme `bozo` avec l'ordonnement $QP||Q$

Nous décrivons ci-dessous le détail du calcul de $prec(QP\chi Q)$:

$$prec(Q) = prec_1(QP) = \emptyset \quad (5.4)$$

$$prec_2(QP) = prec(QP) = \{(Q_1, P_1)\} \quad (5.5)$$

$$prec_1(QP\chi) = prec_2(QP\chi) = prec(QP\chi) = \{(Q_1, P_1), (Q_1, \chi_1), (P_1, \chi_1)\} \quad (5.6)$$

$$prec_1(QP\chi Q) = prec_2(QP\chi Q) = \{(Q_1, P_1), (Q_1, \chi_1), (P_1, \chi_1), (Q_1, Q_2), (\chi_1, Q_2)\} \quad (5.7)$$

$$prec(QP\chi Q) = \{(Q_1, P_1), (Q_1, \chi_1), (P_1, \chi_1), (Q_1, Q_2), (\chi_1, Q_2), (P_1, Q_2)\} \quad (5.8)$$

Nous démarrons avec un ensemble vide (5.4). Nous ajoutons (Q_1, P_1) pour cause de non commutativité de l'attente et de la notification (5.5), puis les couples (Q_1, χ_1) et (P_1, χ_1) qui correspondent au deuxième des trois cas élémentaires pour l'ordre causal (5.6). Viennent ensuite les couples (Q_1, Q_2) et (χ_1, Q_2) grâce aux premier et deuxième cas élémentaires de l'ordre causal (5.7), et enfin (P_1, Q_2) pour cause de fermeture transitive (5.8).

Le couple (Q_1, P_1) est absent de C , et il est permutable car la fermeture transitive de $prec_1(QP)$ donne l'ensemble vide. Par conséquent, les transitions Q_1 et P_1 sont dépendantes et permutable dans $Q_1 P_1 || Q_2$.

5.4.3 Génération des nouveaux ordonnancements

Nous présentons maintenant l'algorithme principal pour la génération des ordonnancements. Cet algorithme s'exécute pour un SSTD donné. Tout d'abord, nous exécutons le SSTD avec un ordonnancement quelconque. Ensuite, nous générons un nouvel ordonnancement chaque fois que nous détectons une nouvelle paire de transitions dépendantes et permutables. La figure 5.6 décrit l'algorithme de façon plus formelle.

```

 $G_S$ (ensemble de contraintes  $C$ ) : //appel initial :  $G_S(\emptyset)$ 
  exécuter le SSTD en respectant les contraintes  $C$  ; (1)
   $u$  = ordonnancement de l'exécution ci-dessus ;
  pour toutes les paires de transitions  $p_i$  et  $q_j$  de  $u$  tel que  $p_i <_u q_j$ 
    et  $(p_i, q_j) \in \mathcal{D} \cap \mathcal{P} | C$  faire : (2)
       $G_S(C \cup \text{"}q_j < p_i\text{"})$  ; //contrainte devant être satisfaite par le nouvel ordonnancement
       $C = C \cup \text{"}p_i < q_j\text{"}$  ; //contrainte déjà satisfaite par l'ordonnancement courant

```

FIG. 5.6 – Algorithme principal pour la génération des ordonnancements

Pour chaque couple de transitions dépendantes et permutables a et b , avec a avant b (formellement $a < b$ et $(a, b) \in \mathcal{D} \cup \mathcal{P}$), l'algorithme ci-dessus doit générer un nouvel ordonnancement. Ce nouvel ordonnancement doit vérifier la contrainte $b < a$. Afin de ne pas générer deux fois le même ordonnancement, il faut que tous les nouveaux ordonnancements générés ultérieurement à partir de ce même ordonnancement courant, respectent la contrainte inverse $a < b$. A chaque génération d'un nouvel ordonnancement, nous divisons ainsi en deux l'ensemble des ordonnancements valides.

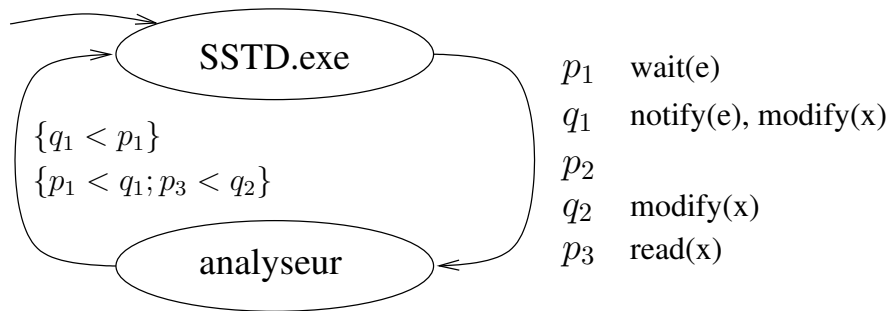
La génération de chaque ordonnancement se fait en deux étapes. D'abord nous générons un ensemble de *contraintes d'ordonnements* que doit vérifier le nouvel ordonnancement. Ensuite, nous générons un ordonnancement qui vérifie cet ensemble de contraintes (ligne (1)). Ces étapes sont détaillées dans les sections qui suivent.

L'algorithme G_S se termine quand il ne détecte plus de nouvelle paire de transitions dépendantes et permutables. Si nous l'exécutons jusqu'à son terme, alors parmi l'ensemble des ordonnancements exécutés, se trouve au moins un représentant de chaque classe d'équivalence. Nous discutons et prouvons cette propriété à la section 5.5.

Nous n'avons pas spécifié l'ordre de parcours des paires de transitions (ligne (2)). Il est important que le parcours de cette boucle se fasse en séquence (et non en parallèle) pour que chaque itération puisse accéder aux modifications de l'ensemble C , effectuées par les itérations précédentes. En revanche, nous verrons que l'ordre de parcours n'a pas d'impact sur la correction de l'algorithme. En pratique, q_j parcourt u de la gauche vers la droite, puis p_i parcourt le préfixe correspondant de la droite vers la gauche. Autrement dit, pour deux paires de transitions (a, b) et (a', b') , (a, b) sera analysé avant (a', b') si $b <_u b'$ ou si $b = b' \wedge a' <_u a$. Cet ordre permet de limiter le nombre d'exécutions en excès par rapport au nombre de classes d'équivalence. Nous parlerons plus en détails de ces exécutions en excès au début du chapitre 7, mais avant nous devons expliquer comment générer effectivement les ordonnancements à partir des ensembles de contraintes.

EXEMPLE 9 — Premier appel à G_S pour l'exemple `bozo`

La figure 5.7 décrit la première itération de l'algorithme G_S pour l'exemple `bozo`. On suppose que la première exécution s'est faite avec l'ordonnancement $p_1q_1p_2q_2p_3$. En analysant la trace


 FIG. 5.7 – Première itération de l'analyse pour l'exemple `bozo`

obtenue, on obtient deux nouveaux ensembles de contraintes. Le premier ensemble de contrainte force la permutation de p_1 et q_1 car ces deux transitions sont permutable et accèdent toutes deux à l'événement `e` de façon non commutative. Cela correspond à la première flèche pointillée du graphe de dépendance dynamique représenté par la figure 5.5. Le deuxième ensemble de contraintes force la permutation de p_3 et q_2 à cause des accès non-commutatifs opérés par ces transitions sur la variable partagée `x` (deuxième flèche pointillée de la figure 5.5). Dans le cas présent, les itérations suivantes n'imposent pas la génération de nouveaux ordonnancements. Au final, nous obtenons donc bien 3 ordonnancements, qui correspondent aux trois cas présentés à la section 4.2. Nous aurions obtenu exactement les mêmes ensembles de contraintes si nous avions considéré l'exemple `bozo++`.

5.4.3.1 Génération de l'ordonnement lors de l'analyse

Le problème consiste maintenant à générer un ordonnancement valide à partir d'un ensemble de contraintes d'ordonnement. La solution initialement envisagée consistait à générer l'ordonnement depuis le contrôleur, c'est-à-dire sans réexécuter le programme mais juste avec les informations sur l'exécution en cours d'analyse.

Ayant calculé l'ordre causal pour un ordonnancement, nous savons que certains autres ordonnancements sont valides aussi. Considérons un ordonnancement de la forme $uavb$ dans lequel nous cherchons à permuter les transitions non-commutative a et b . La section d'ordonnement v se partitionne en trois de la façon suivante :

$$\begin{aligned} v_a &= \{x \in v \mid a \prec x\} \\ v_b &= \{x \in v \mid x \prec b\} \\ v_\perp &= v \setminus (v_a \cup v_b) \end{aligned}$$

Si $v_a \cap v_b$ n'était pas vide, alors les transitions a et b seraient causalement ordonnées, ce qui n'est pas le cas.

En plus de la contrainte $b < a$, nous héritons d'autres contraintes qui doivent aussi être respectées dans le nouvel ordonnancement. Les contraintes héritées servent essentiellement à savoir s'il faut générer un nouvel ordonnancement pour un couple de transitions non-commutatives, ou si cela a déjà été fait.

Plusieurs ordonnancements conviennent, par exemple : uv_bba , uv_bbv_\perp , $uv_bbv_\perp a$ et $uv_\perp v_bba$ mais il en existe de nombreux autres que l'on peut obtenir en mixant les séquences u , v_b et v_\perp . Conserver u comme préfixe est intéressant si l'on veut pouvoir réutiliser les calculs déjà fait pour la

nouvelle exécution, par exemple par clonage du système sous test, ou par sauvegarde de tout ou partie des relations calculées. Dans tous les cas, l'ordonnancement généré doit contenir v_b mais pas v_a . En effet, il se peut que les transitions de v_a ne permettent plus de former un ordonnancement valide ; c'est notamment le cas si la permutation effectuée cause un blocage du processus de a . Les transitions de v_{\perp} peuvent être incluses ou non dans l'ordonnancement généré.

Cette solution coince malheureusement dans un cas. Il se peut que parmi l'ensemble des contraintes héritées, certaines concernent des transitions qui n'ont pas encore été exécutées au moment où l'on découvre une permutation à réaliser. En générant judicieusement les ordonnancements (intuitivement en essayant d'y inclure le plus de transitions possibles), il est seulement possible de rendre ce problème moins fréquent mais pas de le supprimer. Si nous avons réussi à résoudre ce problème, le fonctionnement de l'ordonnanceur aurait consisté à exécuter un ordonnancement pré-calculé, puis une fois son terme atteint à générer librement la suite de l'ordonnancement. Or comme certaines contraintes ne peuvent être prises en compte lors du pré-calcul de l'ordonnancement par l'analyseur, l'ordonnanceur doit tenir compte lui-même des contraintes d'ordonnements fournies.

EXEMPLE 10 — Contraintes sur transitions absentes

On considère l'ordonnancement p_1p_2qr dans lequel $p_1 \prec p_2$ (même processus) et $J = \{(p_1, q), (p_2, r)\}$.

L'analyse de cet ordonnancement résulte en deux ordonnancements fils :

1. à partir de l'ordonnancement p_1p_2q , nous générons un nouvel ordonnancement tel que $q < p_2$, par exemple p_1qp_2 , et nous poursuivons l'exécution courante avec l'hypothèse inverse $p_2 < q$.
2. à partir de l'ordonnancement p_1p_2qr , nous devons générer un nouvel ordonnancement tel que $p_2 < q$ et $r < p_1$. Or la solution décrite ci-dessus permet juste de générer l'ordonnancement rp_1 et donc la contrainte $p_2 < q$ ne peut pas être prise en compte à ce moment là.

5.4.3.2 Génération de l'ordonnancement dynamiquement par l'ordonnanceur

La conclusion du paragraphe précédent est que la génération d'un ordonnancement valide en fonction d'un ensemble de contraintes de la forme $p_i < q_j$ doit pouvoir se faire dynamiquement par l'ordonnanceur. Bien sur, il faut toujours que l'ordonnanceur respecte la spécification de SystemC.

Le principe de la méthode consiste à *geler* les processus, qui, si exécutés une nouvelle fois, violeraient l'une des contraintes imposées.

Définition 14 — Fonction indice

|| Soient u un ordonnancement et p un processus, on note $ind_u(p)$ et on appelle indice de p dans u le nombre d'occurrence de p dans u .

Cette première définition va nous servir pour la définition suivante ; on peut noter que $p_i \in u \Leftrightarrow i \leq ind_u(p)$.

Définition 15 — Ensemble des processus gelés

|| Soit E_u l'ensemble des processus éligibles après exécution d'un ordonnancement u et Ctr l'ensemble des contraintes imposées, l'ensemble G_u des processus gelés dans u est défini ainsi :

$$G_u = \{p \in E \mid \exists (p_i < q_j) \in Ctr, i = ind_u(p) + 1 \wedge j > ind_u(q)\}$$

Et enfin :

Définition 16 — méthode du gel des processus

|| La méthode du gel des processus consiste à n'élire un processus que si il se trouve dans $E_u \setminus G_u$, où u est la partie déjà générée de l'ordonnancement courant.

La consistance est garantie par la définition de l'ensemble des processus gelés ; la complétion est discutée ci-dessous.

EXEMPLE 11 — Application de la méthode du gel des processus

Considérons le système ci-dessous :

- processus p : `cout <<"p1"` ;
- processus q : `cout <<"q1"` ; `yield()` ; `cout <<"q2"` ;

Et générons un ordonnancement vérifiant l'ensemble de contraintes : $\{q_1 < p_1, p_1 < q_2\}$.

- $E_\varepsilon = \{p, q\}$ et $G_\varepsilon = \{p\}$, nous ne pouvons que choisir q . Le processus p est gelé à cause de la première contrainte $q_1 < p_1$.
- $E_q = \{p, q\}$ et $G_q = \{q\}$, nous ne pouvons que choisir p . Exécuter q au pas précédent a dégelé p , mais q se retrouve alors gelé à cause de la deuxième contrainte $p_1 < q_2$.
- $E_{qp} = \{q\}$ et $G_{qp} = \emptyset$, nous ne pouvons que choisir q . Les deux contraintes ne peuvent plus être violées.
- $E_{qpq} = \emptyset$, l'exécution se termine.

Il y a blocage si, à un instant donné, il existe des processus éligibles mais qu'ils sont tous gelés par la méthode ci-dessus. Une question se pose : cette méthode peut-elle mener à une impasse alors qu'une solution existe ? Il existe bien sûr des ensembles de contraintes non satisfaisables comme $\{p_1 < q_1, q_1 < p_1\}$ mais là n'est pas la question. Il s'agit de savoir si un choix d'ordonnancement peut mener dans une voie tel qu'il ne soit ensuite (et seulement ensuite) plus possible de satisfaire les contraintes restantes, comme illustré dans l'exemple ci-dessous.

EXEMPLE 12 — Impasse pour la génération sous contraintes d'un ordonnancement

Processus p : `x = 1` ; `wait(20, SC_NS)` ; `e.notify()` ;
 Processus q : `if (x==1) {wait(e)} else {wait(20, SC_NS)}` ;

Initialement la variable x vaut 0 et les deux processus sont éligibles.

L'objectif est de générer un ordonnancement vérifiant la contrainte $q_2 < p_2$.

Deux constatations s'imposent :

1. La contrainte est vérifiable puisque $q_1 p_1 \chi q_2 p_2$ est un ordonnancement valide.
2. Si p est choisi au premier pas comme l'autorise la méthode du gel des processus, alors il n'est plus possible de satisfaire la contrainte fournie, puisque dans ce cas la transition q_2 sera en attente de la notification de e .

Ainsi, la réponse est que, dans le cas général, la méthode présentée peut mener à une impasse. Heureusement, nous sommes dans un cas bien particulier puisque nous générons les contraintes d'ordonnancement selon des règles qui, en général, permettent d'éviter ces impasses. En effet, nous analysons les ordonnancements de la gauche vers la droite, et à chaque fois que nous rencontrons des transitions non-commutatives, nous ajoutons une contrainte sur leur ordre, et la génération des contraintes ultérieures dépendent indirectement de cette contrainte. Dans le cas de l'exemple ci-dessus, la contrainte $q_2 < p_2$ ne peut être générée qu'en présence de la contrainte $q_1 < p_1$, ce qui empêche tout fourvoiement dans l'impasse présentée.

Il existe cependant des exemples, comme celui ci-dessous, où notre algorithme G_S génère un ensemble de contraintes tel qu'une impasse soit possible. Cependant, il suffit d'analyser les exécutions

stoppées par une impasse de ce type comme les autres exécutions, pour que le résultat final soit correct (c'est-à-dire que l'on a bien atteint tous les états finals possibles). C'est tout au plus un problème d'efficacité puisque les exécutions stoppées par une impasse ne sont pas utiles pour l'ensemble des exécutions finalement obtenu.

EXEMPLE 13 — Ordonnement interrompu

Initialement : le processus Q attend une notification de l'événement e, les autres processus sont éligibles, les variables x et y valent 0.

- Processus P : cout <<'p' ; if (x) cerr <<"Ko";
- Processus Q : cout <<'q' ; x = 19;
- Processus R : cout <<'r' ; if (!y) e.notify();
- Processus S : cout <<'s' ; y = 1;

La séquence des appels récurrents à G_S est donnée par la figure 5.8. Le symbole † y désigne une exécution interrompue, aussi appelée *feuille morte* ; à cet instant, l'ensemble E des processus éligibles contient uniquement le processus P, mais l'élection de celui-ci est interdite par la contrainte d'ordonnement $q < p$.

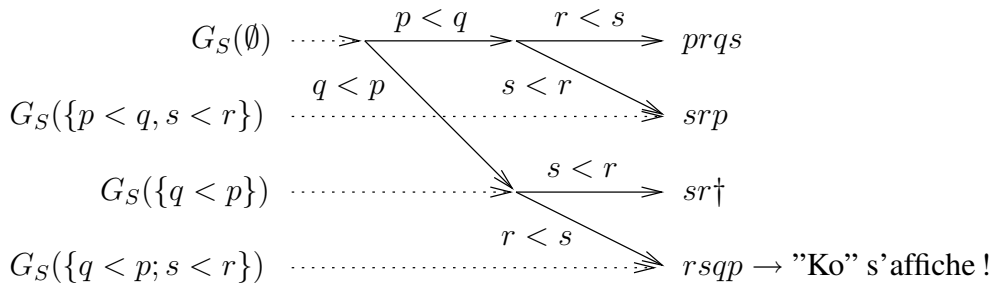


FIG. 5.8 – Arbre des appels à G_S pour l'exemple ci-dessus (les indices des transitions sont omis puisque valant toujours "1")

Il est possible de mixer les deux méthodes : génération du début de l'ordonnement dans l'analyseur, puis complétion en fonction des contraintes dans l'ordonneur. Cela est notamment utile si l'on souhaite avoir un préfixe commun le plus long possible entre l'ordonnement père et l'ordonnement généré.

5.5 Mise en application pour la validation

Nous avons fini de décrire la génération des ordonnancements. Nous allons maintenant discuter des propriétés vérifiées par le jeu d'ordonnements généré, et des conséquences pratiques pour la validation du modèle sous test.

5.5.1 Propriété principale

Exécuter l'algorithme G_S jusqu'à son terme, en générant les ordonnancements lors de la simulation via la méthode du gel des processus, fournit un jeu d'ordonnements qui vérifie la propriété suivante :

Propriété 5 Soit M un SSTD. On note A_M l'ensemble de tous les ordonnancements valides, et E_M un ensemble quelconque d'ordonnancements générés par G_S . On a alors :

$$\forall u \in A_M, \exists v \in E_M, u \equiv v$$

Informellement, on obtient au moins un représentant de chaque classe d'équivalence.

Étant donné que l'algorithme G_S n'est pas déterministe puisque certaines portions d'ordonnancements sont générées librement, il est possible d'obtenir des ensembles E_M différents d'une exécution de G_S à l'autre, mais ils vérifient tous cette propriété. En particulier, G_S ne visite pas toujours le même nombre d'impasses pour un SSTD donné. Nous discutons en détails des impasses et des exécutions redondantes à la section 7.1.

Une conséquence directe est que l'on visite tous les états finaux accessibles : $\forall u \in A_M, \exists v \in E_M, F_M(u) = F_M(v)$, puisque $u \equiv v \Rightarrow F_M(u) = F_M(v)$. Nous verrons à la sous-section suivante comment cela permet la validation du SSTD.

5.5.1.1 Arbre des contraintes d'ordonnancements

Pour prouver cette propriété, nous allons montrer comment, étant donné un ordonnancement valide de A_M , on peut trouver un élément de E_M qui lui est équivalent. L'idée consiste à compléter l'algorithme G_S de façon à ce qu'il engendre un arbre binaire qui représente l'ordre de génération des contraintes et des ordonnancements. Chaque arête de l'arbre engendré est étiquetée par une contrainte d'ordonnement, et chaque feuille correspond à un élément de E_M . Pour tout ordonnancement, il est possible de parcourir cet arbre de sa racine à une feuille de façon à ce que toutes les contraintes associées aux arêtes empruntées, soient respectées par l'ordonnement considéré.

Considérons par exemple l'ordonnement $rqsp$ de l'exemple 5.4.3.2 et l'arbre représenté par la figure 5.8. Deux arêtes partent de la racine du graphe : l'une étiquetée par $p < q$ et l'autre par la contrainte inverse $q < p$. Dans $rqsp$, q est avant p donc on suit l'arête descendante. On regarde ensuite l'ordre de r et s : r est avant s dans $rqsp$ donc on suit de nouveau l'arête descendante, ce qui nous mène à l'ordonnement $rsqp$ qui a été effectivement exécuté. Les ordonnancements $rqsp$ et $rsqp$ sont bel et bien équivalents puisque q et s sont indépendants (ils n'accèdent pas aux mêmes objets).

La version complétée de l'algorithme G_S est donnée par la figure 5.9.

```

 $G_S$ (ensemble de contraintes  $C$ , noeud  $n$ ) : //appel initial :  $G_S(\emptyset, racine)$ 
exécuter le SSTD en respectant les contraintes  $C$  ;
 $u$  = ordonnancement de l'exécution ci-dessus ;
pour toutes les paires de transitions  $p_i$  et  $q_j$  de  $u$  tel que  $p_i <_u q_j$ 
    et  $(p_i, q_j) \in \mathcal{D} \cap \mathcal{P} \setminus C$  faire :
         $(n, n') = n \rightarrow \text{créer\_arêtes\_et\_noeuds}("p_i < q_j", "q_j < p_i")$ 
         $G_S(C \cup "q_j < p_i", n')$ ;
         $C = C \cup "p_i < q_j"$ ;
 $n \rightarrow \text{ordonnement} = u$  //ici,  $n$  est une feuille
    
```

FIG. 5.9 – Algorithme principal G_S complété pour la génération de l'arbre des ordonnancements

L'appel à la méthode " $n \rightarrow \text{créer_arêtes_et_noeuds}(c_d, c_b)$ " qui prend deux contraintes en argument modifie le noeud n ainsi :

1. un nouveau noeud n_d (d comme “droite”) est créé et une arête étiquetée par c_d reliant n à n_d est ajoutée au graphe ;
2. un nouveau noeud n_b (b comme “bas”) est créé et une arête étiquetée par c_b reliant n à n_b est ajoutée au graphe ;
3. la méthode retourne le couple de noeuds (n_d, n_b) .

Nous n’avons fait qu’ajouter des instructions qui ne modifient pas la façon dont les ordonnancements sont générés. Cette version renvoie donc les mêmes jeux d’ordonnements que la version précédente.

Nous notons T_M l’arbre généré (T comme *Tree*), et $C(u)$ les contraintes associées à un ordonnancement u de E_M , c’est-à-dire l’ensemble des contraintes situées sur les arêtes reliant la racine de T_M à la feuille d’étiquette u . Par construction, u vérifie les contraintes $C(u)$ (formellement : $\forall u \in E_M, u \models C(u)$). En effet, chaque création d’arête s’accompagne d’un ajout de la contrainte associée à l’ensemble de contraintes courant (arête “droite”), ou à l’ensemble généré (arête “descendante”).

De plus, toujours par construction, pour tout ordonnancement v et noeud n ayant deux arêtes $n \xrightarrow{c_d} n_d$ et $n \xrightarrow{c_b} n_b$, l’ordonnancement v vérifie au moins c_d ou c_b puisque selon la version complétée de G_S , ces deux contraintes sont toujours opposées. L’ordonnancement v ne peut vérifier simultanément les deux contraintes que si les deux transitions correspondantes sont absentes de v (si $c_d = “p_i < q_j”$ et donc $c_b = “q_j < p_i”$, v ne vérifie à la fois c_b et c_d que si p a été exécuté au plus $i - 1$ fois, et q $j - 1$ fois). On note $f(n, v)$ le noeud n_b si $v \models c_b, n_d$ sinon.

Cette fonction intermédiaire f permet de définir une fonction e_M (e comme *équivalent*) qui à tout ordonnancement de A_M va associer une feuille de T_M et donc un ordonnancement de E_M qui a été exécuté par G_S .

Définition 17 — Fonction de classement des ordonnancements

Soit r la racine de T_M et v un ordonnancement de A_M . On a $e_M(v) = e_M(v, r)$ et pour tout noeud n :

$$\begin{aligned}
 e_M(v, n) &= \text{l'ordonnancement associé à } n \text{ si } n \text{ est une feuille} \\
 &= e_M(v, f(n, v)) \quad \text{sinon}
 \end{aligned}$$

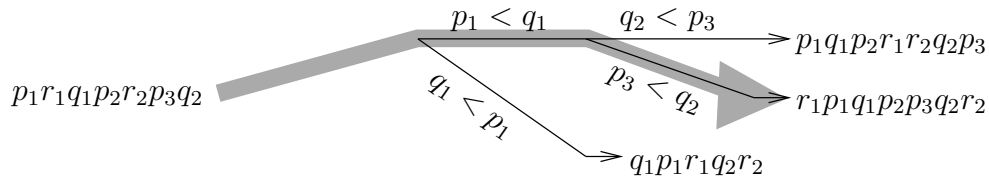


FIG. 5.10 – Arbre des contraintes d’ordonnement générées pour l’exemple `bozo++`, et classement d’un ordonnancement : $e_M(p_1r_1q_1p_2r_2p_3q_2) = r_1p_1q_1p_2p_3q_2r_2$.

5.5.1.2 Preuve de la propriété principale

D’après la définition de la fonction de classement e_M , si $u = e_M(v)$ alors $v \models C(u)$. La preuve de la propriété principale 5 se résume donc à montrer que :

$$\forall v \in A_M, v \models C(e_M(v)) \Rightarrow v \equiv e_M(v)$$

Autrement dit : $C(e_M(v))$ est un jeu de contraintes complet, selon la définition ci-dessous.

Définition 18 — Jeu complet de contraintes

Un jeu de contraintes d'ordonnements C est dit complet si et seulement si :

$$\forall u, v, u \models C \wedge v \models C \Rightarrow u \equiv v$$

Preuve : On note u l'ordonnement $e_M(v)$. On sait que $v \models C(u)$ et il nous faut montrer que $u \equiv v$. Pour cela, nous allons permuter les transitions de u jusqu'à obtenir v .

On note $u_0 = u$ et pour tout u_n , on calcule $u_{n+1} \equiv u_n$ de la façon suivante. Tant que $u_n \neq v$, l'ordonnement v s'écrit sous la forme $v = w_n p_i v'_n$ où w_n est le préfixe commun à u_n et v .

Pour pouvoir écrire u_n sous la forme $u_n = w_n u'_n p_i u''_n$, il faut d'abord montrer que $p_i \in u_n$. La transition p_i est éligible dans l'état $F_M(w)$ (c'est-à-dire que le processus p est éligible pour la i -ème fois), par conséquent il y a deux hypothèses :

1. soit p_i est présent dans u ;
2. soit p_i est encore éligible dans l'état $F_M(u)$, puisque dans le langage considéré un processus ne peut pas être désactivé.

Si p_i est éligible dans l'état $F_M(u_n)$ et donc aussi à la fin de l'exécution de u , cela implique que la transition p_i a été gelée par une contrainte d'ordonnement de la forme $q_j < p_i$ et que q_j est absent de u et u_n . Or v doit aussi vérifier cette contrainte, ce qui implique que $q_j \in w_n$ et donc $q_j \in u_n$ ce qui est contradictoire. Par conséquent seule la première hypothèse est correcte.

Le but est maintenant de transformer $u_n = w_n u'_n p_i u''_n$ en $u_{n+1} = w_n p_i u'_n u''_n$ par permutations successives de p_i avec les transitions de u'_n . On note q_j la transition qui précède immédiatement p_i . La transition p_i est éligible à la place de q_j car déjà éligible dans l'état $F_M(w_n)$, par conséquent $(q_j, p_i) \in \mathcal{P}$. D'après l'algorithme de G_S et comme $u_n \equiv u$, $(q_j, p_i) \in \mathcal{D}$ impliquerait que " $q_j < p_i$ " $\in C(u)$, et donc $w_n p_i \models "q_j < p_i"$ or ceci est contradictoire avec $q_j \notin w_n$. En conséquence, q_j et p_i sont indépendantes et ainsi la permutation est possible en restant dans la même classe d'équivalence. En appliquant le même raisonnement pour chaque permutation de p_i , on obtient l'ordonnement u_{n+1} souhaité.

Chaque fois que l'on calcule u_{n+1} à partir de u_n , on obtient un nouveau préfixe w_{n+1} strictement plus grand que son prédécesseur w_n . Par conséquent, en un nombre fini N d'itérations (borné par la longueur de v), on obtient un ordonnement u_N tel que $u \equiv u_N$ et $v = u_N s$. Si v correspond à une exécution complète, alors aucun processus n'est éligible dans l'état $F_M(v) = F_M(u_N)$ et donc la portion d'ordonnement s est vide et $u \equiv v$. ■

Dans cette preuve, on a utilisé le fait qu'un processus éligible ne peut pas être désactivé, c'est-à-dire qu'il reste nécessairement éligible jusqu'à être élu. Il serait intéressant de trouver les adaptations à apporter à l'algorithme et à sa preuve pour éliminer cette contrainte.

5.5.2 Conséquences pour la validation

Pour la validation, nous allons utiliser le fait que l'on visite au moins une fois chaque état final. L'idée est donc de faire en sorte que la présence d'une erreur lors d'une exécution mène à état final reconnaissable.

5.5.2.1 Détection des inter-blocages

Nous allons nous intéresser d'abord aux interblocages (ou “*deadlock*”). Il y a interblocage quand tous les processus sont en attente alors que l'exécution du test n'a pas atteint sa fin normale.

Les interblocages constituent déjà des états finaux. La seule difficulté consiste à distinguer les interblocages des états finaux normaux. On trouve deux solutions dans les études de cas industrielles :

- à la fin du code de chaque composant maître, on ajoute une transition spéciale qui indique que ce composant a bien atteint la fin de ce qu'il avait à faire.
- à la fin de l'exécution du SSTD, on vérifie que les données obtenues en sortie sont bien complètes.

La première solution nécessite de veiller à ce qu'il n'y ait pas un interblocage au niveau des composants esclaves après la fin de l'exécution des composants maîtres. Dans la plupart des cas, on peut programmer le composant maître de façon à ce qu'il vérifie que les composants esclaves ont fini leur travail avant de lancer la transition spéciale qui marque la fin de sa propre exécution.

Dans certains cas, la deuxième solution peut être plus simple à mettre en œuvre. Elle nécessite de pouvoir récupérer les données générées à la toute fin du traitement dans un format convenable pour des comparaisons ultérieures. Une sauvegarde de l'état de la mémoire (“*dump*”) n'est pas toujours suffisante. Par exemple si le but est d'afficher une image, il faut aussi vérifier que l'écran a bien affiché ce qui était dans la mémoire vidéo. Si on utilise cette solution, le SSTD est exécuté une première fois, les données générées sont vérifiées éventuellement manuellement, puis ces données servent ensuite de modèle de référence pour les exécutions suivantes.

5.5.2.2 Détection des erreurs locales

Nous appelons *erreurs locales* les erreurs que l'on peut détecter en n'observant qu'un seul processus. La vérification de propriétés de sûreté peut se ramener au problème de l'accessibilité locale [AS87]. D'après la propriété principale, l'algorithme G_S nous fournit au moins un représentant de chaque classe d'équivalence. Par ailleurs, nous avons dit à la section 5.3.4 que les transitions de deux ordonnancements sont égales deux à deux, c'est-à-dire qu'elles exécutent la même section de code et accèdent aux mêmes valeurs. Les transitions d'un processus étant complètement ordonnées au sein d'une classe d'équivalence, un observateur extérieur qui ne regarde qu'un seul processus ne peut donc pas différencier deux ordonnancements équivalents. En conséquence, si un de ces *observateurs locaux* détecte une erreur pour un ordonnancement valide de A_M , alors il la détectera aussi sur les ordonnancements équivalents à celui-ci, et donc sur un ordonnancement de E_M exécuté par G_S .

5.5.2.3 Détection des erreurs non-locales

Les choses se compliquent légèrement lorsqu'il s'agit de détecter des erreurs qui dépendent du comportement de plusieurs processus. Nous allons regarder cela sur un exemple minimal.

EXEMPLE 14 — Observateur non-local

x (respectivement y) est une variable locale du processus p (respectivement q). Chaque processus ne comporte qu'une transition, et celle-ci est réduite à une affectation.

- $p : x = 1;$
- $q : y = 1;$

Par ailleurs, un observateur o est présent et exécute le code suivant :

- $o : \text{erreur} = (y==1) \ \&\& \ (x==0); \ \text{assert}(!\text{erreur});$

L'observateur vérifie que y ne vaut pas 1 tant que x vaut 0.

Le système constitué des deux processus p et q n'a que deux ordonnancements valides : pq (correct) et qp (erroné). Ceux-ci sont équivalents. Lancer l'algorithme G_S sur ce système renvoie un singleton, par exemple $E_M = \{pq\}$. L'erreur visée par l'observateur o n'est pas présente lors de l'exécution de cet ordonnancement pq , donc même si on exécute l'observateur o entre chaque transition, le bug restera caché.

La solution consiste à inclure l'observateur au système que l'on fournit à G_S . On considère désormais le système constitué des processus p , q et o , où le processus o est initialement éligible et exécute le même code que l'observateur précédent. Le nouveau système autorise 6 ordonnancements différents : opq , poq et pqo qui sont corrects ; oqp , qop et qpo qui sont incorrects. L'erreur n'est détectée pas l'observateur que si on exécute l'ordonnancement qop , mais la transition o est dépendante et permutable avec p et q car elle lit des variables que p et q modifient. Ainsi, G_S va générer plus d'ordonnements. La figure 5.11 représente une exécution possible de G_S sur ce système à trois processus. En conséquence de la propriété principale, l'ensemble E_M contient nécessairement l'ordonnancement qop qui permet de détecter le bug.

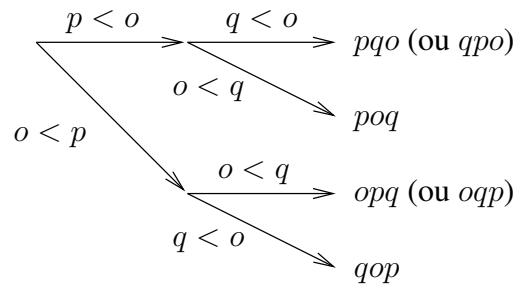


FIG. 5.11 – Arbre des ordonnancements générés par G_S pour l'exemple 5.5.2.3

Nous sommes contents d'avoir détecté l'erreur, mais il est dommage d'avoir exécuté 4 fois un système qui initialement n'avait que deux ordonnancements valides. Pour éviter ce regrettable inconvénient, il est possible d'inclure l'observateur au système initial de façon plus judicieuse. En effet, il suffit d'exécuter le code de l'observateur après avoir modifié la variable y . On obtient donc le système suivant :

```
- p : x = 1;
- q + o : y = 1; erreur = (y==1) && (x==0); assert(!erreur);
```

Ainsi, le nombre d'ordonnements n'est pas augmenté par rapport à la version sans observateur et G_S génère désormais l'ensemble $E_M = \{p(q+o), (q+o)p\}$. Le bug est détecté lors de l'exécution de $(q+o)p$.

En résumé, il est possible de vérifier des propriétés globales avec cette méthode, mais il y a encore des travaux à mener pour le faire efficacement.

Chapitre 6

Implantation

Sommaire

6.1	Architecture	82
6.2	Ordonnanceur interactif	83
6.2.1	Interface avec l'implantation OSCI	83
6.2.2	Fonctionnalités du nouvel ordonnanceur	84
6.3	Enregistrement des traces d'exécutions	85
6.3.1	Contenu et format	85
6.3.2	Modification du noyau SystemC	87
6.3.3	Instrumentation du modèle	87
6.4	Calcul des dépendances pour une exécution	92
6.4.1	Analyse syntaxique du fichier XML	92
6.4.2	Les structures de données	93
6.4.3	Calcul de l'ordre partiel	93
6.5	Génération de l'ensemble des ordonnancements	96
6.6	Outils annexes	97
6.6.1	Génération des graphiques de dépendances	97
6.6.2	Enregistrement d'une trace détaillée	99
6.6.3	Arbres des contraintes d'ordonnement	101
6.7	Conclusion	101

Nous avons présenté le principe des algorithmes de façon assez abstraite au chapitre précédent. Nous allons maintenant voir comment ils peuvent s'implanter et s'intégrer à une chaîne de validation complète. La chaîne de validation a été réalisée sous forme de plusieurs petits programmes indépendants, afin d'améliorer la modularité de l'outil complet, et de faciliter la réutilisation de certaines pièces.

La problématique est double. D'une part, le prototype doit permettre de valider des exemples de taille réelle malgré la complexité intrinsèquement exponentielle de l'algorithme global. D'autre part, le temps passé par l'utilisateur doit aussi être raisonnable par rapport au bénéfice escompté. Le temps passé par l'utilisateur se divise en trois : apprentissage, préparation du SSTD et lancement de la validation, et enfin analyse des résultats.

La plupart des outils développés lors de cette thèse ont été initialement baptisés avec un nom commençant par "rv", ce qui signifie ici *Runtime Verification*, en français : *vérification à la volée*. En effet, l'outil analyse des exécutions pour y trouver des problèmes cachés, ce qui est bien l'idée

centrale des techniques de vérification à la volée. Depuis, d'autres techniques ont été introduites dans notre chaîne d'outils, mais, pour raisons historiques, le préfixe "rv" est resté.

L'architecture globale de la chaîne de validation développée pendant cette thèse est décrite par la première section. Avant toute chose, il faut remplacer l'ordonnanceur par défaut de l'implantation OSCI par une version interactive, ce qui est décrit par la section 6.2. Afin de récupérer les informations nécessaires au calcul des dépendances, le code de SystemC ainsi que celui du système sous test doit être instrumenté (section 6.3.3). Ensuite, il est possible de calculer les dépendances pour une exécution donnée (section 6.4), puis de générer un jeu complet d'ordonnements (section 6.5). Enfin, la génération des graphiques des dépendances statiques et dynamiques est décrite à la section 6.6.1.

6.1 Architecture

L'architecture est décrite par la figure 6.1. Le flot démarre d'un modèle TLM écrit en SystemC, puis via plusieurs outils et formats intermédiaires, nous obtenons d'une part des ordonnancements menant à d'éventuelles erreurs, et d'autre part des graphiques représentant les communications possibles et réelles.

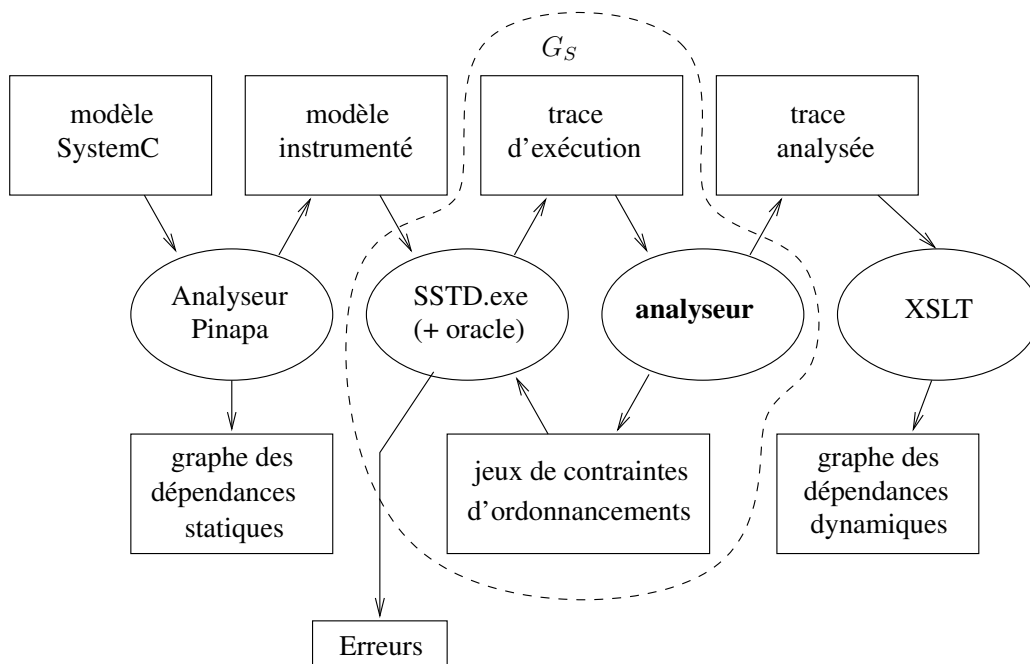


FIG. 6.1 – Schéma global de l'architecture. Les noeuds rectangulaires représentent les données ; les noeuds ovales représentent les exécuteurs.

Le cœur du flot est constitué de la boucle passant par l'exécuteur du SSTD, qui correspond à la première ligne de G_S (cf figure 5.6), et par l'analyseur, qui correspond à la boucle interne de G_S . D'autres outils sont nécessaires en amont et en aval pour préparer le modèle, et mettre les résultats sous une forme plus exploitable par les humains.

Nous allons détailler le rôle et le fonctionnement de chaque partie du flot dans les sections suivantes. Chacun des traitements a nécessité soit des modifications d'une application existante, soit un développement complet.

6.2 Ordonnanceur interactif

Nous avons vu à la section 4.1 que la spécification d'un ordonnanceur SystemC n'est pas déterministe. Cependant l'implantation OSCI, presque exclusivement utilisée dans le contexte des modèles abstraits, fixe un choix particulier d'ordonnement pour chaque SSTD. Pour essayer d'autres ordonnancements valides, il est nécessaire de disposer d'un ordonnanceur "interactif".

Intuitivement, le cœur d'un ordonnanceur reçoit ou calcule à chaque pas d'exécution une liste des processus éligibles, puis choisit un élément de cette liste. Pour tester le modèle avec d'autres ordonnancements, il faut remplacer le code qui effectue ce choix.

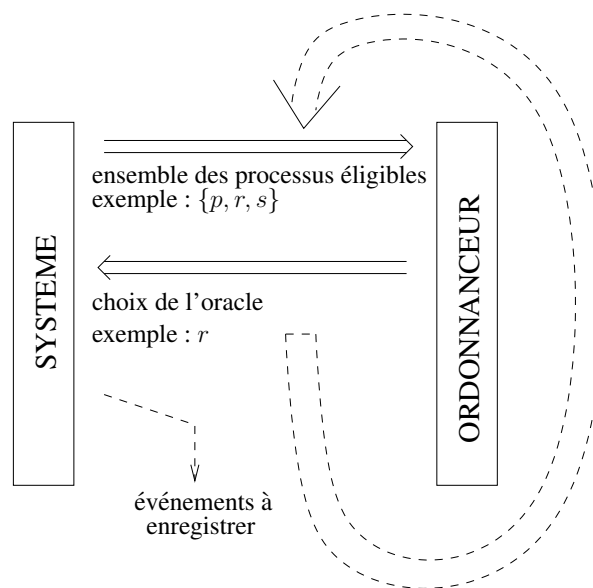


FIG. 6.2 – Schéma de fonctionnement d'un ordonnanceur interactif.

6.2.1 Interface avec l'implantation OSCI

Il est bien sûr hors de question de développer un nouveau noyau de simulateur SystemC en entier. Notre solution consiste donc à modifier le noyau de l'implantation OSCI, dont les sources sont publiques, pour obtenir les fonctionnalités voulues.

L'implantation OSCI utilise 4 files pour stocker les processus éligibles, 2 pour les méthodes (SC_METHOD) et 2 pour les threads (SC_THREAD). Initialement, seule une des files des méthodes n'est pas vide. L'ordonnanceur parcourt cette file et exécute successivement chacun de ses éléments. Les processus activés (c'est-à-dire rendus éligible) par des notifications immédiates sont alors placés dans la seconde file, comme représenté par la figure 6.3. L'ordonnanceur traite ensuite les threads de la même façon. Si les deux files qui récoltent les nouveaux processus éligibles sont vides, l'ordonnanceur passe à la phase de mise à jour (changement de δ -cycle); sinon le rôle des files est inversé et l'ordonnanceur reprend l'étape précédente. Cela revient à découper chaque δ -cycle en une séquence de micro-cycles pendant lesquels un processus s'exécute au plus une fois. Ce découpage est un choix d'implantation et n'est pas imposé par la spécification. Ces files sont accessibles via les instructions `mh = pop_runnable_method()` et `push_runnable_method(mh)` pour les méthodes, ou `th = pop_runnable_thread()` et `push_runnable_thread(th)` pour les threads.

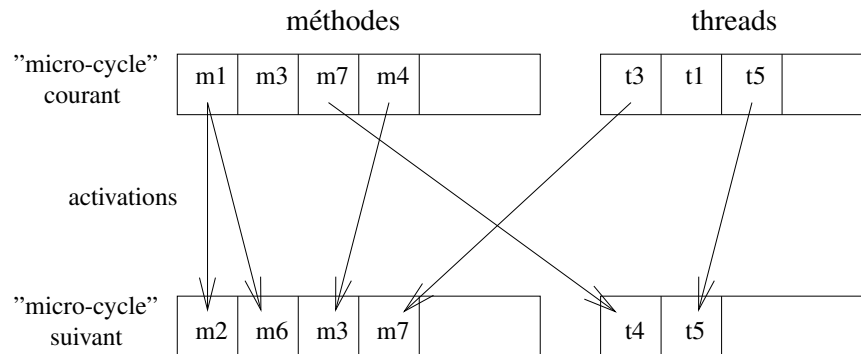


FIG. 6.3 – Files des processus éligibles et micro-cycles. Dans cet exemple, la méthode m1 active les méthodes m2 et m6, la méthode m3 n’active aucun autre processus, m7 active le thread t4, ... etc.

Nous nous autorisons à modifier les traitements, mais pas à modifier les structures de données existantes car cela rendrait nos modifications trop intrusives, augmentant les risques d’incompatibilité entre versions et compliquant la maintenance de notre version modifiée. Nous avons retenu la méthode la plus simple, même si elle n’est probablement pas la plus efficace. Au début d’un micro-cycle, nous demandons d’abord à un “oracle” de choisir un processus parmi une liste de processus éligibles, puis nous parcourons les deux files : pour chaque processus, s’il s’agit du processus élu alors nous l’exécutons normalement, sinon nous transférons directement le processus dans la file des processus du même éligible au micro-cycle suivant.

6.2.2 Fonctionnalités du nouvel ordonnanceur

L’alternative la plus simple à l’ordonnanceur par défaut consiste à effectuer le choix dans la liste des processus éligibles de façon aléatoire. Il est aussi utile de pouvoir guider la simulation au clavier, ainsi que de pouvoir enregistrer des ordonnancements pour les réexécuter ultérieurement. De plus, pour implanter l’algorithme G_S , l’ordonnanceur doit être capable de gérer des contraintes d’ordonnancements.

Toutes ces fonctions ont été implantées, sans difficulté technique particulière. Le fonctionnement du nouvel ordonnanceur est contrôlable via les options de la ligne de commande, l’exécutable étant celui obtenu en compilant le modèle et en le liant avec le SystemC modifié.

EXEMPLE 15 — Utilisation de l’ordonnanceur modifié

Les commandes tapées par l’utilisateur sont en gras.

```

- mode interactif :
> ./bozo
SystemC 2.1.v1 --- Sep 19 2006 17:07:54
Copyright (c) 1996-2005 by all Contributors
ALL RIGHTS RESERVED
RVS List of eligible processes:
RVS Id: 0
RVS Id: 1
RVS Our Choice: 1 'l' comme liste : correspondance pid ↔ nom
pour chaque processus SystemC
0 TOP.P

```

```

1      TOP.Q
RVS    List of eligible processes:
RVS    Id:    0
RVS    Id:    1
RVS    Our Choice: 0           choix du processus de pid 0 (TOP.P)
RVS    List of eligible processes:
RVS    Id:    1
RVS    Our Choice: 1           choix du processus de pid 1 (TOP.Q)
RVS    List of eligible processes:
RVS    Id:    0
RVS    Our Choice: D           pour terminer la simulation avec l'ordonnan-
                                     ceur par défaut
RVS    Scheduler Oracle: use OSCI scheduler until end of
simulation
Ok
- mode aléatoire :
> ./bozo -rv-random -rv-seed=42
[...] Ok
> ./bozo -rv-random -rv-seed=13
[...] Ko
- chargement d'un fichier de contraintes :
> cat leaf.ctr           les processus SystemC sont représentés par leur pid
1_1>0_1                 signifie (TOP.Q)1 après (TOP.P)1
1_2>0_3                 signifie (TOP.Q)2 après (TOP.P)3
> bozo -rv-random -rv-ctr=leaf.ctr
[...] Ko

```

Dans la suite, nous utiliserons principalement le mode aléatoire avec contraintes.

6.3 Enregistrement des traces d'exécutions

La génération des contraintes d'ordonnancement est basée sur l'analyse des traces d'exécutions. Dans cette section, nous décrivons comment les enregistrer.

6.3.1 Contenu et format

La trace d'une exécution doit contenir :

- le découpage en cycles temporels et δ -cycles,
- la liste des pas d'exécutions de chaque cycle avec l'identifiant du processus élu,
- pour chaque pas d'exécution, l'ensemble des actions de communication exécutées par le processus,
- pour chaque notification, les processus qui ont été activés par celle-ci,
- enfin, une légende pour la correspondance entre les identifiants numériques et les noms réels.

Nous avons choisi d'enregistrer les traces dans un format XML. Les raisons de ce choix sont : la possibilité pour un humain de lire directement la trace puisque c'est du texte, la facilité pour lire la trace depuis un programme et l'existence de nombreux outils dédiés. Le format exact d'une trace d'exécution est spécifié par une *Définition de Type de Document* (DTD), disponible avec les sources

du prototype. Le format est relativement simple puisque chaque élément apparaît toujours avec la même profondeur.

Les différents objets mentionnés dans la trace sont identifiés par des entiers. L'implantation OSCI associe déjà un identifiant numérique (*pid*) à chaque processus, en se basant sur leur ordre d'instanciation qui est constant d'une exécution à l'autre. L'analyse d'une trace ne nécessite que les identifiants numériques, mais pour les utilisateurs humains, il est souhaitable de disposer de la correspondance entre les identifiants et les noms réels. Cette correspondance est décrite par la *légende*. En général, nous utilisons l'inclusion XML (balise `<xi:include>`) pour ne pas devoir recopier la légende dans chaque trace d'exécution.

EXEMPLE 16 — Trace complète d'une exécution du programme `bozo`

```
<?xml version="1.0"?>
<run>
  <legend>
    <processes total="2">
      <process pid="0" name="P" module="TOP" />
      <process pid="1" name="Q" module="TOP" />
    </processes>
    <events total="1">
      <event eid="0" name="e" module="TOP" />
    </events>
    <variables total="1">
      <var vid="0" name="x" module="TOP" />
    </variables>
  </legend>
  <chi date="0">
    <delta num="1">
      <step num="1" pid="0">
        <ev_wait eid="0" />
      </step>
      <step num="2" pid="1">
        <ev_notify eid="0">
          <ev_enable pid="0" />
        </ev_notify>
        <var_write vid="0" modified="true" />
        <wait_time duration="10" />
      </step>
      <step num="3" pid="0">
        <wait_time duration="10" />
      </step>
    </delta>
  </chi>
  <chi date="10">
    <delta num="1">
      <step num="5" pid="1">
        <var_write vid="0" modified="true" />
      </step>
    </delta>
  </chi>
</run>
```

```
    <die />
  </step>
  <step num="6" pid="0">
    <var_read vid="0" />
    <die />
  </step>
</delta>
</chi>
</run>
```

—

Si l'on souhaite par exemple retrouver l'ordonnancement utilisé, il suffit d'extraire la séquence des attributs `pid` des éléments `step`, et de consulter la légende pour obtenir la séquence de noms de processus voulue.

6.3.2 Modification du noyau SystemC

L'enregistrement se fait d'une part grâce à des ajouts dans le simulateur SystemC, et d'autre part par instrumentation du modèle lui-même. La modification du simulateur SystemC est nettement plus simple que l'instrumentation du modèle, essentiellement parce que la modification du simulateur SystemC est faite une fois pour toute, et que celui-ci doit de toutes façon être modifié pour disposer d'un ordonnanceur interactif. Par conséquent, tout ce qui peut être fait depuis le simulateur SystemC, sera fait ainsi.

Nous avons tout d'abord défini une classe `rvs_recorder` pour centraliser les événements à enregistrer. L'enregistreur, instance de cette classe, se charge de configurer la sortie, de stocker quelques informations temporaires, de mettre les données au format XML et d'appliquer quelques filtres dont nous verrons plus loin l'utilité.

L'enregistrement des changements de cycle, des élections de processus et des notifications ou attentes d'événement est techniquement simple. Il suffit de trouver l'endroit approprié dans les sources de l'implantation OSCI et d'y ajouter une ligne du type : `recorder->elect(mh->proc_id)`.

Contrairement à d'autres structures comme les modules et les signaux, la classe `sc_event` n'est pas une classe fille de la classe `sc_object`. Par conséquent, les événements ne disposent pas d'un attribut `nom`. Les événements SystemC ne disposent pas non plus d'identifiant numérique par défaut mais cela est facile à ajouter. Il faut associer les identifiants numériques à des noms de variables du programme, mais malheureusement les noms des variables ne sont plus accessibles lors de l'exécution. Une idée de solution est de profiter de l'étape d'analyse statique nécessaire à l'instrumentation pour établir la correspondance entre le code et les objets dynamiques.

6.3.3 Instrumentation du modèle

Les variables partagées jouent un rôle important pour la synchronisation des processus. Une variable est considérée comme partagée à partir du moment où elle peut être accédée par deux processus distincts. Certaines variables partagées servent juste pour le contrôle ; par exemple, un booléen peut permettre de signaler si un processus est prêt ou non à traiter une requête. D'autres servent aux transferts de données ; leur taille est généralement plus élevée.

6.3.3.1 Architecture détaillée

Dans le cadre d'une thèse dans la même équipe sur la transformation automatique de modèles TLM en modèles formels basés sur des automates synchrones, Matthieu Moy a développé Pinapa : un analyseur syntaxique (ou "front-end") pour SystemC, qui appartient désormais au domaine des logiciels libres [SCI05, MMMC05b]. Pinapa utilise `g++` pour l'analyse syntaxique en elle-même, et fournit donc des arbres abstraits (AST) au même format que le front-end `g++`. L'idée consiste à utiliser cet outil pour analyser le code du modèle afin de détecter quelles variables sont partagées, puis insérer dans les sources le code chargé de l'enregistrement des accès. La réalisation de cet "instrumenteur", nommé `sc2rvs`, a été confiée à Frédéric Saunier, ingénieur Silicomp.

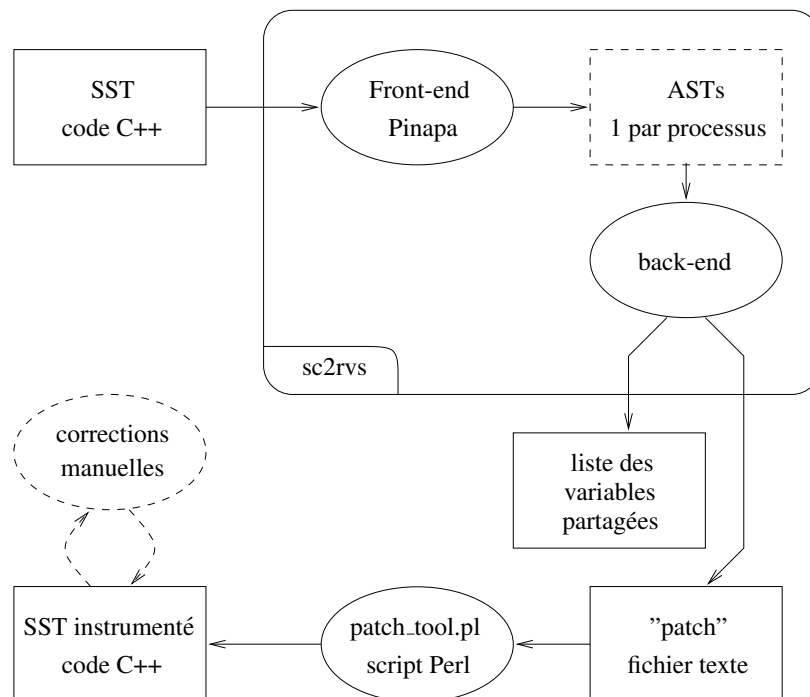


FIG. 6.4 – Architecture détaillée pour l'étape d'instrumentation.

La figure 6.4 détaille le flot des traitements qui doit mener au modèle instrumenté (SST). L'instrumentation ne dépend pas des données. Tout d'abord, le front-end Pinapa nous permet d'accéder à l'arbre abstrait de chaque processus SystemC (`SC_METHOD` et `SC_THREAD`). Le back-end parcourt ces arbres et détecte les accès en lecture ou écriture aux variables. Si une variable n'est accédée que depuis un seul processus, alors il n'y a rien à faire, sinon il faut ajouter au niveau de chaque accès une instruction permettant d'enregistrer l'accès. Le code à ajouter peut être `rvs_get_recorder()->read(42)` ou `rvs_get_recorder()->write(42)` selon qu'il s'agit d'une lecture ou d'une écriture ; l'argument numérique est l'identifiant de la variable. Dans le cas d'un accès à un tableau, par exemple `T[i]`, l'identifiant utilisé est `id(T) + i`.

Une idée consisterait alors à ajouter des noeuds dans l'arbre abstrait puis re-générer du code C++ à partir de l'AST, mais malheureusement la mise en œuvre de cette idée n'est pas possible car nous ne disposons pas d'une fonction capable de re-générer du C++ à partir de l'AST, et nous n'avons pas les moyens suffisants pour la créer nous-mêmes. L'autre solution, actuellement mise en œuvre, consiste à générer un fichier texte décrivant les modifications à appliquer aux sources. Ce "patch" est

une liste de couples numéro de ligne - code à ajouter. Un script Perl ad-hoc permet ensuite d'appliquer les modifications décrites aux sources.

6.3.3.2 Intégration du code d'instrumentation dans le code source

Cette dernière étape qui consiste à insérer des nouvelles lignes dans le code source pose problème. En effet, selon comment le code source est formaté, l'insertion de nouvelles lignes peut causer des erreurs de syntaxe, voir, dans certains cas plus rares, modifier la sémantique.

- Si on se contente d'insérer la nouvelle ligne devant la ligne ciblée, la sémantique est modifiée si la ligne cible constitue le corps d'un `if` sans accolade.

```
if (C)
    rvs_get_recorder()->write(42) //ligne insérée
    x = 12; //ligne cible
```

- Le problème ci-dessus peut être résolu en ajoutant des accolades autour de la ligne insérée et de la ligne cible, comme ci-dessous.

```
if (C)
    { rvs_get_recorder()->write(42) //ligne insérée
      x = 12; //ligne cible
    } //ligne insérée
```

- Cependant, les accolades ne doivent pas être insérées dans les cas suivants :
 - l'expression cible est la condition d'un `if` ou d'une boucle `while` ou `for` (à noter que dans le cas du `while`, la ligne d'instrumentation doit aussi être insérée à la fin du corps de la boucle);
 - la ligne cible comporte une déclaration, par exemple `int y = x;`, car la portée de la variable serait modifiée.
 - Dans le cas d'une structure `if` écrite sur une seule ligne, par exemple `if (C) I;`, si l'on souhaite instrumenter l'instruction du corps `I`, il faut couper la ligne après la condition `C` pour pouvoir insérer le nouveau code sans erreur.

Cela est juste un aperçu des problèmes que l'on a rencontrés en instrumentant des modèles fournis par STMicroelectronics. D'autres problèmes surviennent lorsque des expressions sont écrites sur plusieurs lignes. Certains cas sont détectables et traitables automatiquement, d'autres non (sauf à refaire une analyse syntaxique complète pour l'insertion du code, mais ce n'est pas envisageable en l'état). Des règles strictes sur le formatage du code pourraient résoudre de nombreux problèmes, mais celles-ci n'existent pas. Des retouches manuelles restent donc nécessaires.

6.3.3.3 Identifications des vecteurs de communications

La méthode ci-dessus suppose que l'on associe à chaque vecteur de communication (variables partagées et événements SystemC) un identifiant numérique. Cela permet de générer lors de cette phase une légende décrivant la correspondance entre les objets et leur identifiants numériques. Néanmoins, cette technique a plusieurs inconvénients et requiert quelques compléments.

Tout d'abord, l'association des identifiants est faite statiquement pour un modèle complet. Il faut donc modifier les identifiants si on intègre un composant déjà instrumenté à un nouveau modèle. Or comme l'instrumentation n'est pas entièrement automatisée, il est indispensable de ne pas devoir refaire un même travail plusieurs fois. La solution de ce problème est la suivante : nous associons un identifiant statique aux objets qui est interne au composant, puis nous calculons une *base* dynamique

pour chaque composant lors de l'instanciation des modules. Cela donne des lignes d'instrumentation de la forme `rvs_get_recorder()->read(module_base + 12)`. La légende globale du modèle est alors remplacée par des légendes par composants.

L'autre problème vient des objets instanciés dynamiquement, ou accédés via des pointeurs. La variable, où plus précisément la case mémoire, n'est pas connue lors de l'instrumentation. Dans ce cas, on fournit directement l'adresse en mémoire de l'objet à l'enregistreur, qui se charge alors de normaliser les identifiants. Les lignes d'instrumentation sont alors de la forme `rvs_get_recorder()->read(&x)`. La mise en œuvre de cette technique est simple, mais il n'est plus possible de générer une légende pour les identifiants. Les outils de générations d'ordonnements qui suivent n'ont besoin de connaître que la plage des identifiants. Pour rétablir le lien avec le code source, un outil annexe sera décrit à la sous-section 6.6.2.

6.3.3.4 Enregistrement des accès aux composants mémoires

Les modèles de SoCs contiennent généralement un ou plusieurs composants mémoires. Ces mémoires constituent une immense réserve de "variables partagées" accessibles par tous les processus ou presque. Deux approches extrêmes sont possibles. Une mémoire peut être considérée comme un seul objet partagé ; dans ce cas deux transitions seront considérées comme dépendantes dès qu'elles accèdent à la même mémoire. Cette méthode demande peu de ressources mais constitue une abstraction conservatrice mais trop forte. A l'extrême inverse, chaque octet de la mémoire peut être considéré comme une variable partagée à part entière. Dans ce cas, aucune abstraction n'est faite, mais les outils qui suivent ont peu de chance de survivre avec des milliers, voir des millions, d'objets partagés. Le même problème se pose pour tout les tableaux de grande dimension.

En observant les accès mémoires des exemples réels, nous avons constaté que les transitions des modèles TLM n'accèdent généralement qu'à un nombre limité de plages mémoires. Ces plages mémoires sont de taille variable, et elles peuvent être lues ou écrites en une où plusieurs fois selon la granularité des transactions. Notre objectif est d'éviter les abstractions, sans générer des traces d'exécutions trop lourdes, et donc trop coûteuses à enregistrer puis à analyser. L'idée est d'enregistrer directement les accès mémoires sous forme d'une liste de plages. Par exemple :

```
<mem_access mid="0" start="4194304" size="4096"/>
<mem_access mid="0" start="120832" size="11296"/>
<mem_access mid="0" start="16384" size="22656"/>
```

Pour simplifier, nous ne distinguons pas ici les lectures des écritures. Comme les accès à une plage se font souvent en plusieurs fois, il faut rassembler les accès contigus, et n'inscrire les accès à la mémoire qu'à la fin de la transition. Pour cela, notre enregistreur utilise un conteneur associatif ordonné pour chaque composant mémoire. L'implantation utilise la classe `map` de la STL, mais elle pourrait aussi utiliser directement des arbres binaires de recherche. Ces conteneurs associent les débuts des plages avec leur taille. L'algorithme d'insertion d'une nouvelle plage est réalisé de telle sorte que l'invariant suivant soit vérifié : pour toutes paires de plages $(start_1, size_1)$ et $(start_2, size_2)$, on a $start_1 + size_1 + 1 < start_2$. En simplifiant, l'algorithme d'insertion d'une plage $(start, size)$ recherche d'abord $start$ puis $start + size$, puis insère une nouvelle plage en l'absence de collision ou modifie les bornes existantes, et enfin supprime les éventuelles plages incluses dans la nouvelle.

6.3.3.5 Approches alternatives

L'approche décrite ci-dessus nécessite d'ajouter du code au niveau de chaque accès à un objet partagé. Ces ajouts de code étant sources de difficultés, il est souhaitable de minimiser le nombre de

ces ajouts. Une idée, proposée et développée par Jérôme Cornet (doctorant Verimag - STMicroelectronics), consiste à remplacer les variables partagées par des *sondes*.

Concrètement, il s'agit de remplacer les déclarations de la forme `T x;`, par `probe<T> x`, pour toutes les variables partagées. La classe template `probe<T>` définit deux attributs : l'identifiant de type entier positif et défini dynamiquement par le constructeur, et la valeur de type `T`. Ensuite, cette classe redéfinit les opérateurs courants de façon à détecter les accès. Dans le code de chaque redéfinition, se trouvent une instruction d'enregistrement et une instruction pour modifier la valeur comme l'aurait fait le code normal. Voici deux exemples de redéfinition, l'un pour l'affectation, l'autre pour la conversion vers le type d'origine `T`. Ce deuxième opérateur est appelé implicitement lors de chaque lecture.

```
template<typename T>
probe<T> & probe<T>::operator=(const T & t)
{
    rvs_get_recorder()->write(id,value!=t);
    value = t;
    return *this;
}
template<typename T>
probe<T>::operator T () // casting operator
{
    rvs_get_recorder()->read(id);
    return value;
}
```

Cette approche a de nombreuses limitations. Elle n'est notamment applicable qu'aux types de bases, et fonctionne mal avec les appels de fonction. Cependant, elle s'est montrée très efficace lors des études de cas. Elle permet d'instrumenter manuellement et assez rapidement des modèles petits et moyens. En utilisant l'outil d'instrumentation pour faire une partie du travail, dont le repérage des variables partagées, il est possible d'instrumenter des modèles de grande taille en un temps raisonnable (c'est-à-dire moins d'une journée de travail).

Une autre solution, radicalement différente, consisterait à détecter et filtrer tous les accès à la mémoire (du simulateur) via un outil comme `gdb`. Il est à priori possible de demander à `gdb` d'interrompre la simulation à chaque accès mémoire et d'exécuter une fonction chargée de filtrer et enregistrer l'accès. Nous n'avons pas mis cette approche en œuvre. Il est probable que la simulation du modèle soit très fortement ralentie.

6.3.3.6 Modèles TLM avec sections de code source indisponibles

Une méthode basée sur une instrumentation ou une interprétation du binaire aurait l'avantage de permettre une validation de type *boite noire*. Cependant, même avec les approches basées sur des modifications du code source, il est possible d'instrumenter des modèles dont certaines portions de code source ne sont pas disponibles. En effet, les sections de code qui n'accèdent pas directement à des variables partagées, n'ont pas besoin d'être instrumentées. Cela est généralement le cas des bibliothèques logicielles de traitement de données, qui peuvent donc être ignorées durant la phase d'instrumentation.

Il arrive aussi que le logiciel embarqué ne soit disponible que sous forme binaire. Deux cas sont alors à distinguer.

1. Soit il s'agit du binaire pour le processeur du SoC (*compilation croisée*), et dans ce cas il sera interprété par un ISS qui pourra intercepter toutes les communications voulues.
2. Soit il s'agit d'un binaire temporaire pour le processeur du simulateur (*compilation native*), et dans ce cas il faut regarder l'implantation du *wrapper* (i.e. le module qui charge ce binaire et gère son interface), pour voir s'il y a des variables partagées.

6.4 Calcul des dépendances pour une exécution

Nous allons maintenant présenter l'outil d'analyse des traces d'exécution. Cet outil, nommé *rvsc*, prend en entrée une trace d'exécution et éventuellement un ensemble de contraintes d'ordonnement précédemment générées, et doit rendre en sortie :

- la liste des couples de transitions dépendantes et permutable (format XML) ;
- pour chacun de ces couples, un ensemble de contraintes d'ordonnement forçant la permutation des transitions visées ;
- facultativement un jeu *complet* de contraintes pour l'exécution analysée, c'est-à-dire une liste de contraintes finalement respectées par cette exécution, et qui garantissent que toute autre exécution les satisfaisant est équivalente.

La figure 6.5 représente l'architecture interne de l'analyseur *rvsc*. Il est composé de trois grandes parties : un analyseur XML, un module *checker* chargé du calcul de l'ordre partiel, et un ensemble de sous-modules gérant les types de structures contrôlées. Cette dernière séparation est conçue pour faciliter l'ajout de nouvelles structures de données, mais ces sous-modules communiquent étroitement avec le module principal.

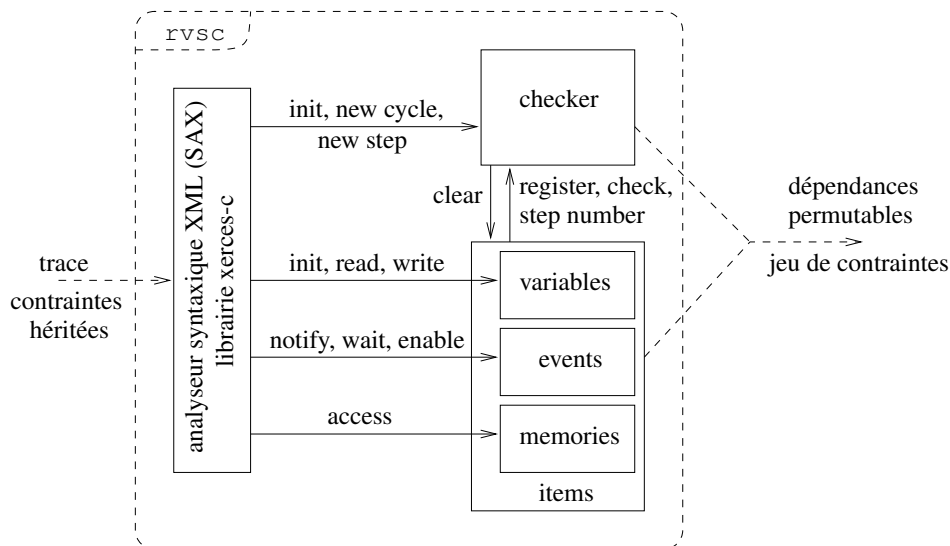


FIG. 6.5 – Architecture détaillée de l'analyseur.

6.4.1 Analyse syntaxique du fichier XML

Il y a peu de choses à dire sur ce module, si ce n'est qu'il utilise le mode événementiel de type SAX, et non une approche hiérarchique de type DOM. C'est-à-dire que l'analyseur XML ne construit

pas une représentation en mémoire de l'arbre complet reconnu, mais se contente d'appeler les fonctions requises pour chaque balise XML reconnue. Il ne consomme donc pas plus de mémoire que nécessaire. Ce module est réalisé avec la librairie Xerces-C++ développée par Apache.

A noter que de légères modifications sur l'enregistreur permettraient de le brancher directement sur l'analyseur, sans passer par un fichier XML ; le calcul des dépendances se ferait alors à la volée lors de la simulation. Dans le cadre de cette thèse, nous avons préféré disposer de briques bien disjointes.

6.4.2 Les structures de données

Le module d'analyse est le cœur de cet outil, bien qu'il se présente sous une forme qui fait plutôt penser à une librairie. Il enregistre diverses données lui permettant de tenir à jour un grand tableau représentant l'ordre partiel calculé. L'implantation utilise un grand nombre de tableaux, plus exactement des vecteurs de la STL. Le rôle de chacune de ces structures de données est décrit par la table 6.1.

L'enregistrement de la liste des contraintes d'ordonnement est gérée par une classe spécifique, aussi utilisée par l'ordonneur interactif. La gestion des accès aux mémoires du modèle se fait aussi avec une structure spécifique, mais différente de celle utilisée par l'enregistreur car il nous faut ici associer des numéros de pas d'exécution à chaque plage mémoire. Nous utilisons pour cela un conteneur associatif ordonné associant des bornes d'intervalles à des numéros de pas d'exécution, comme illustré par l'exemple ci-dessous.

EXEMPLE 17 — Gestion des accès aux mémoires

On souhaite stocker les informations ci-dessous :

- la plage $[0, 6]$ a été accédée par le pas 11
- la plage $[10, 12]$ a été accédée par le pas 22
- la plage $[13, 42]$ a été accédée par le pas 33

Nous stockons les couples : $((0 \mapsto 11), (7 \mapsto 0), (10 \mapsto 22), (13 \mapsto 33), (43 \mapsto 0))$.

Si l'on souhaite maintenant connaître le dernier accès effectué à l'adresse 12, nous recherchons la borne inférieure de 12 via la fonction `lower_bound` fournie par la STL, et nous obtenons le numéro de pas 22 via la clef 10.

—
Afin d'économiser de la mémoire lors de l'analyse, il est possible de réinitialiser toutes les structures de données lors des changements de cycle. En effet, les permutations ne concernent toujours que les transitions du cycle courant. En pratique, un seuil permet de déterminer si une ré-initialisation serait rentable.

6.4.3 Calcul de l'ordre partiel

Chacune des fonctions appelées par l'analyseur XML doit mettre à jour les structures de données, et vérifier la non-permutabilité en cas d'actions non-commutatives. Voici, pour exemple, le code de la fonction appelée dans le cas d'une écriture, avec des commentaires détaillés.

```
// Le paramètre v est l'identifiant numérique de la variable partagée concernée.
void rv_var::write(unsigned v, bool modified) {
    // La valeur de la variable a-t-elle été modifiée par cette écriture ?
    if (modified) {
        // Une modification n'est pas commutative avec un autre accès.
```

Nom	Profil	Description
process	step → process	process [s] ==p ⇔ le processus de pid p a été exécuté au pas numéro s
last	process → step	last [p] ==s ⇔ le processus p n'a pas été exécuté depuis le pas s
wake	process → step	wake [p] ==s ⇔ le processus p a été activé au pas s
lastpred	step × process → step	lastpred [s1] [p] ==s2 ⇔ pour tout pas s3 ≤ s2 tel que process [s3] ==p, s3 est causalement avant s1
ctr	(process × occ × process × occ)*	jeu complet de contraintes pour la portion déjà analysée de la trace sujette
Step	process × occ → step	Step [p] [n] ==s ⇔ le processus p a été exécuté pour la n-ème fois au pas s
StepOcc	step → occ	StepOcc [s] ==n ⇔ Step [process [s]] [i] ==s
ProcOcc	process → occ	ProcOcc [p] ==n ⇔ le processus p a été exécuté n fois
varM	variable → step	varM [v] =s ⇔ la variable v a été modifiée pour la dernière fois au pas s
varA	variable × process → step	varA [v] [p] =s ⇔ la variable v a été accédée pour la dernière fois au pas s par le processus p
evN	event × process → step	evN [e] [p] =s ⇔ l'événement e a été notifié pour la dernière fois au pas s par le processus p
evWC	event × process → step	evWC [e] [p] =s ⇔ l'événement e a été attendu ou "reçu" pour la dernière fois au pas s par le processus p
Wevents	process → event*	liste des événements attendus (OR_LIST) par le processus p
memA	≈ address range → step*	structure spécifique pour déterminer les dernières transitions qui ont affectées une plage d'adresses d'une mémoire donnée

TAB. 6.1 – Structures de données utilisées par l'analysteur.

6.4. Calcul des dépendances pour une exécution

```
// Nous demandons donc une vérification avec les accès précédents.
checker->check(varA[v], "var", v);
// La fonction check se charge de la génération éventuelle d'une ou plusieurs contraintes.
varM[v] = checker->current_step; // Nous enregistrons la modification.
}
// Nous enregistrons l'accès.
varA[v][checker->current_process] = checker->current_step;
}
```

Les éléments `varM[v]` et `varA[v]` ne sont pas du même type : l'un est un singleton alors que l'autre est un tableau indexé par les identifiants de processus. Cette différence provient du fait que les optimisations possibles dépendent du type d'action. Ces optimisations sont basées sur le principe suivant : *étant donné une action α d'une transition a , nous avons besoin d'un ensemble de transitions E tel que pour toute transition b précédant strictement a , et contenant une action β non-commutative avec α , soit b est dans E , soit b est causalement avant une transition de E .* Pour bloquer toutes les dépendances permutable, il suffit alors de générer une contrainte d'ordonnancement pour chaque élément de E qui est permutable avec a . Les transitions permutable absentes de cet ensemble E seront traitées par les appels récursifs à G_S .

Toutes les modifications étant causalement ordonnées, il suffit de mémoriser la dernière pour chaque variable. Autrement dit, si α est une modification, alors un singleton est suffisant pour E . En revanche, deux accès quelconques, par exemple deux lectures, ne sont pas nécessairement causalement ordonnés. La solution consiste alors à mémoriser le dernier accès effectué par chaque processus, en profitant du fait que toutes les transitions d'un même processus sont causalement ordonnées. La fonction `check` effectue alors la vérification pour chacun des éléments de cette liste, par exemple pour chaque élément de `varA[v]`.

La plupart des actions de communication créent des dépendances permutable, et ressemblent à celle ci-dessus. Entre deux changements de cycle, seules les activations de processus (balise `<enable ...>` dans les traces) issues des notifications permettent de forcer l'ordre de deux transitions. Nous donnons ci-dessous les extraits correspondants : d'abord la fonction `enable` correspondant à la balise du même nom, puis la fonction `set_pred` appelée à chaque nouvelle transition, par la fonction `add_step` et avec les arguments `(wake[p], current_step)`.

```
// p est l'identifiant du processus réveillé.
void rv_ev::enable(unsigned p) {
    // La prochaine transition de p sera causalement après la transition courante.
    checker->set_waker(p, checker->current_step);
    // Nous appelons la fonction catch_ sur chaque événement attendu par p,
    // afin de détecter d'éventuelles dépendances entre notifications (cf 5.4.1.2).
    for_each(Wevents[p].begin(), Wevents[p].end(),
        std::bind1st(std::mem_fun(&rv_ev::catch_), this));
    Wevents[p].clear();
}
```

La fonction `set_pred` permet d'ajouter une contrainte à l'ordre causal. Il faut faire en sorte de conserver l'invariant $a < b \Leftrightarrow a \leq \text{lastpred}[b][\text{process}[a]]$, qui permet de savoir en temps

constant si deux transitions sont permutable. Il n'est possible d'ajouter une contrainte $a \prec b$ que si b est la transition courante.

*// s est le numéro d'une transition que l'on déclare, en appelant cette fonction, causalement avant
// la transition courante.*

```
void rv_checker::set_pred(unsigned s) {
    transform(lastpred[current_step].begin(),
             lastpred[current_step].end(), // 1ère source
             lastpred[s].begin(), // 2ème source
             lastpred[current_step].begin(), // destination
             ptr_fun<[...]>(max)); // fonction appelée sur chaque paire
    assert(is_pred(s, current_step)); désormais : s < current_step
}
```

L'invariant est bien conservé. Le tableau `lastpred` est noté lp avant l'appel et lp' après ; le pas courant est noté c . Soit a le numéro d'une transition et p son processus :

- si $a \prec s$, alors $a \leq lp[s][p] \leq \max(lp[s][p], lp[c][p]) = lp'[c][p]$;
- si $a \prec c$, alors $a \leq lp[c][p] \leq \max(lp[s][p], lp[c][p]) = lp'[c][p]$;
- sinon $a > lp[s][p] \wedge a > lp[c][p] \Rightarrow a > \max(lp[s][p], lp[c][p]) = lp'[c][p]$.

Enfin, la fonction `check` vérifie si la transition courante p_i est permutable avec une transition q_j précédente et dépendante ; son nom est surchargé afin de pouvoir traiter toute une liste de transitions en un seul appel. C'est la fonction la plus importante mais son implantation ne présente pas de difficulté technique : évaluer la permutable se fait en temps constant grâce au tableau `lastpred`, et si la permutation est possible, il suffit d'ajouter la contrainte " $q_j < p_i$ " à la structure de données `Ctrl`, et à l'ordre partiel via la fonction `set_pred` ci-dessus. Au passage, un jeu de contrainte est écrit dans un fichier texte et servira de contraintes d'ordonnancement initiales pour une future exécution.

6.5 Génération de l'ensemble des ordonnancements

Nous avons décrit ci-dessus deux applications indépendantes :

- le système sous-test et ses données, instrumenté et compilé avec un noyau SystemC modifié ;
- l'analyseur de traces d'exécution qui génère des jeux de contraintes ciblant des comportements pertinents à tester.

Un script, développé en Perl, se charge de faire le lien entre ces deux exécutables. Celui-ci, nommé tout simplement `rvs`, appelle alternativement le SSTD et l'analyseur. Il commence par simuler une première fois le système avec un jeu de contraintes vide, puis appelle l'analyseur sur la trace générée. Les noms des fichiers de contraintes générés sont stockés dans une pile. Le script dépile ensuite ces fichiers un par un en appelant successivement le SSTD puis l'analyseur avec les bonnes options. L'exécution du script se termine quand la pile est vide. Les résultats des simulations et des analyses sont stockées dans des sous-répertoires (un par simulation).

Une option "`-check=cmd`" permet d'exécuter une commande supplémentaire à la fin de chaque couple simulation-analyse. Elle est généralement utilisée avec une commande chargée de vérifier les sorties générées par la simulation.

6.6 Outils annexes

Les divers fichiers générés ne sont guère agréables à lire par un utilisateur humain. Il s'agit souvent de longs fichiers textes ou XML qui utilisent essentiellement des identifiants numériques. Les outils décrits dans cette dernière section ont pour objectif de présenter les synchronisations sous une forme plus lisibles, et de fournir des informations supplémentaires aidant à identifier les causes des erreurs détectées.

6.6.1 Génération des graphiques de dépendances

La première idée consiste à générer automatiquement les graphiques que l'on a défini à la section 5.3.3, à savoir les graphes de dépendances statiques et dynamiques.

6.6.1.1 Dépendances statiques

Selon la description faite au chapitre précédent, les noeuds du graphe des dépendances statiques représentent les processus, et ses arcs relient les processus qui peuvent accéder à un même objet partagé. Les informations représentées proviennent du code source des processus. Le principe de base est le suivant : si une variable x apparaît dans le code d'un processus et d'un autre (ou d'une des fonctions qu'ils appellent), alors on ajoute une arête étiquetée par x entre ces deux processus.

Cela nécessite le même parcours de code que l'instrumentation. Il a donc été facile de compléter l'outil d'instrumentation basé sur Pinapa pour extraire les informations nécessaires.

Nous, c'est-à-dire Frédéric Saunier et moi, avons manqué de temps pour parfaire et tenir à jour cette extension. Il reste donc plusieurs limitations, dont les principales demanderaient des analyses statiques plus complexes.

- Actuellement, aucune analyse n'est faite sur les adresses d'objets C++. Seuls les objets existants statiquement (ou au moins à la fin de la phase d'élaboration de SystemC, là où Pinapa nous rend la main) sont donc représentés.
- Nous ne faisons pas non plus d'analyse sur les adresses SystemC. Par conséquent, lorsque l'on rencontre un code générant une transaction (par exemple `port.write(addr, value)`), nous ne savons pas quel composant esclave sera appelé. Plutôt que de faire une abstraction très large consistant à supposer l'existence de toutes les possibilités, nous avons choisi de considérer les fonctions qui reçoivent les transactions comme des pseudo-processus. Cela rend d'une certaine façon cette analyse compositionnelle, puisque nous obtenons ainsi des graphes indépendants pour chaque composant SystemC-TLM. Une autre analyse, utilisant notamment la *carte des plages mémoires* (ou *memory map*), serait nécessaire pour obtenir le graphe global.

Mis à part que nous souhaitons représenter les processus et pseudo-processus d'un même composant cote à cote, nous n'avons pas de contraintes sur le placement. Nous avons donc opté pour le format *dot* et l'outil *dotty* [Kou94], pour l'enregistrement et l'affichage de ces graphes.

6.6.1.2 Dépendances dynamiques

Un exemple est donné par la figure 5.5 du chapitre précédent. Contrairement aux dépendances statiques, l'obtention des informations n'est pas difficile puisqu'elles sont déjà toutes présentes dans les traces XML générées lors de la simulation, et complétées par l'analyseur avec les couples de dépendances permutable.

La position des noeuds, qui représentent tous des transitions, est entièrement fixée :

- l'abscisse est fournie par le numéro du pas,

- l'ordonnée est fournie par le numéro du processus.

Il est par conséquent inutile d'utiliser un outil de placement automatique comme nous l'avons fait pour le graphe des dépendances statiques.

Le choix le plus important porte sur le format de sortie. Une première version utilisait le format du bien connu outil de dessin *Xfig*. Il s'agit d'un format texte où tout est représenté par des nombres, peu lisible à l'oeil nu mais aisé à générer. L'un des avantages est qu'il est possible de l'exporter vers de nombreux autres formats. Cependant, *Xfig* n'est pas disponible pour toutes les systèmes d'exploitation.

Pour la version actuelle nous nous sommes portés vers un format plus moderne, à savoir le *SVG* (*Scalable Vector Graphics*). Ce format fait l'objet d'une spécification détaillée, écrite par le W3C (*World Wide Web Consortium*) [Con03]. Il est basé sur XML et permet de décrire des graphiques vectoriels, ce qu'il nous faut. Voici ci-dessous un extrait représentant une flèche bleue :

```
<line x1="300" x2="300" y1="170" y2="110"
      stroke="blue" stroke-width="1"
      marker-end="url(#blueEndArrow)"/>
```

Ce langage est de plus en plus supporté¹ par les navigateurs Web récents, et devraient donc être lisible sur tous les OS.

La génération du graphe consiste donc à transformer un fichier XML en un autre XML. Le langage *XSL* [Con99b] a été conçu spécialement pour ce genre de problème. Il s'agit d'un langage déclaratif. En gros, on demande que tel type d'élément soit remplacé par tel motif, sans fixer de contrainte sur l'ordre de parcours. Les *variables XSL* sont en réalité des constantes puisqu'il n'est pas possible de modifier leur valeur. Le principal avantage qui en découle est qu'il suffit d'écrire ce qui est spécifique à notre application. Notamment, il est inutile de se préoccuper de l'analyse syntaxique puisque l'interpréteur s'en charge.

Le bilan de notre utilisation d'une *XSLT* est mitigée. Le langage nous a permis de décrire la transformation souhaitée avec peu de commandes ; le programme complet ne prend qu'à peine plus de 200 lignes. Cependant, le code est à la fois très verbeux et peu agréable à lire. Cela est en partie dû au fait que *XSL* est aussi un dialecte XML, mais aussi à un manque d'expressivité. Certaines choses apparemment complexes s'écrivent facilement, grâce notamment à la possibilité d'utiliser des requêtes *Xpath* [Con99a], alors que des calculs arithmétiques simples sont long à écrire. Le code ci-dessous illustre cela ; son rôle est de créer un attribut *y1* valant $(p + 1) \times 100 \pm 30$ suivant le cas. Cela aurait pu s'écrire plus simplement si l'on avait disposé d'une expression conditionnelle (comme le (C) ?E:F de C) dans les attributs *select*.

//@pid (attribut) et \$p (variable) sont deux identifiants de processus

```
<xsl:choose>
  <xsl:when test="@pid>$p"> // lire @pid>$p
    <xsl:attribute name="y1">
      <xsl:value-of select="($p+1)*100+30"/>
    </xsl:attribute> [...]
  </xsl:when>
  <xsl:otherwise>
```

¹Nous aurions préféré pouvoir écrire "déjà supporté", mais les limitations actuelles encore trop nombreuses nous en empêchent.

```
<xsl:attribute name="y1">
  <xsl:value-of select="($p+1)*100-30"/>
</xsl:attribute> [...]
</xsl:otherwise>
</xsl:choose>
```

Ce problème devrait pouvoir être résolu dans l'avenir grâce à des syntaxes alternatives plus "humaines". Notre autre inquiétude concerne le passage à l'échelle et est plus fondamentale. En effet, pouvoir faire des hypothèses sur l'ordre de parcours permettrait des optimisations pour le traitement de gros fichiers, par exemple en remplaçant les nombreux appels à la fonction *count* par des compteurs que nous incrémenterions au fur et à mesure. Nous arrivons là à un choix entre les avantages de la programmation impérative et de la déclarative.

6.6.2 Enregistrement d'une trace détaillée

Il est temps de traiter un problème que nous avons reporté jusque là, à savoir refaire le lien entre les identifiants numériques et le code source. Considérons l'extrait de trace d'exécution ci-dessous :

```
<step num="101" pid="0">
  <var_read vid="316" />
  <var_write vid="316" modified="true" />
  <ev_notify eid="1" />
  <wait_time duration="10" />
</step>
```

Nous souhaitons retrouver plusieurs informations :

- Quel est le processus d'identifiant 0? Quelle est la variable d'identifiant 316? Quel est l'événement d'identifiant 1?
- A quel endroit du code commence cette transition?
- A quel endroit du code ont eu lieu ces actions?
- Le cas échéant, par quelles fonctions la fonction courante a-t-elle été appelée?

Retrouver le nom du processus est facile ; il y a tout ce qu'il faut dans l'API SystemC pour cela. Pour les autres objets, cela peut être plus compliqué selon la technique d'instrumentation utilisée. Les points suivants font appel à des informations absentes de la trace d'exécution. Le dernier point nécessite de pouvoir retrouver la pile d'exécution, ce que l'on ne peut pas obtenir par de simples techniques d'instrumentation. Par ailleurs, il nous faut réutiliser au maximum des outils existants afin de limiter les développements nécessaires.

L'idée consiste à ré-exécuter le SSTD avec les mêmes arguments dans le débogueur *GDB*, et à utiliser les possibilités de programmation de cet outil pour obtenir les informations requises. Le principe est décrit par la figure 6.6.

Du côté du SSTD, nous utilisons le même binaire que lors de la génération des ordonnancements. Les arguments sont transmis via le fichier de commandes. En dehors de l'instruction de lancement, ce fichier contient des commandes *GDB* qui sont toutes construites sur le même modèle. Voici ci-dessous le code pour les écritures sur variables partagées :

```
break rvs_gdb_write
```

→ *Nous plaçons un point d'arrêt sur cette fonction (vide, mais appelée à l'endroit propice).*

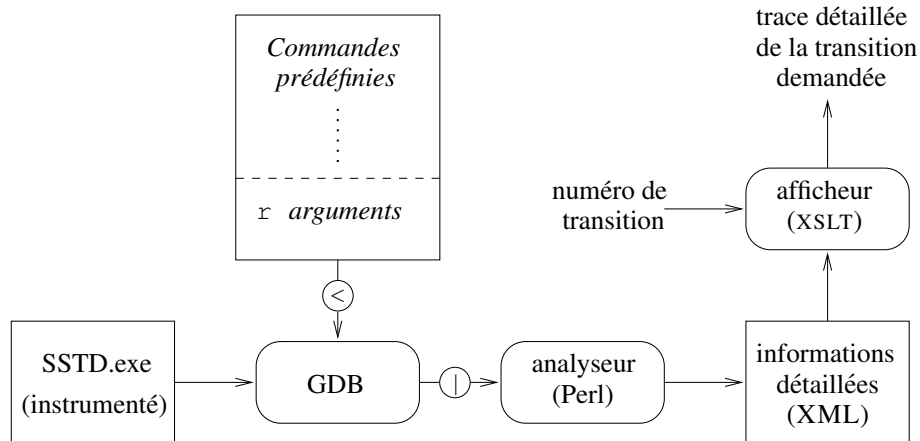


FIG. 6.6 – Architecture pour l'obtention de traces détaillées.

```
commands
```

→ *Ce mot clef indique les commandes devant être exécutées à chaque activation de ce point d'arrêt.*

```
silent
```

→ *Nous désactivons les affichages inutiles.*

```
p "+++iwrite+++"
```

→ *Cette affichage facilitera l'analyse de la trace obtenue.*

```
p var_id
```

→ *Nous récupérons l'identifiant de la variable.*

```
frame 3
```

→ *Nous récupérons via cette commande notre position dans le code, et la ligne de code courante.*

(Pour obtenir toute la pile d'exécution, il suffit d'utiliser bt à la place.)

```
cont
```

→ *l'exécution repart automatiquement.*

```
end
```

Si une classe n'est pas utilisée dans un programme SystemC, il se peut que les méthodes correspondantes soient absentes de l'exécutable final, car ignorées par `ld` lors de la liaison des bibliothèques statiques. Or placer un point d'arrêt sur une fonction absente provoquerait une erreur dans `GDB`. Un script shell, nommé `rvig`, se charge de placer, dans le fichier de commandes, uniquement les sections pertinentes. Il utilise pour cela la commande Unix `nm` permettant de vérifier la présence des symboles. Ce même script se charge du lancement de `GDB` et de l'appel à l'analyseur Perl `rvip`.

Un script Perl se charge ensuite d'analyser les sorties générées par `GDB` pour les filtrer et les stocker dans un fichier structuré par des balises XML. Un petit afficheur, nommé `rvl`, permet ensuite d'extraire au format texte les informations d'un pas d'exécution donné. L'enregistrement d'une trace détaillée est nettement plus lent ($\approx \times 10$) qu'une exécution normale. Par conséquent, nous ne les générons qu'à la demande. Voici un extrait du résultat final correspondant à l'exemple ci-dessus :

```
Action: write (variable 316)
Where:  in video_display_pv::write (this=0x81f1010, [...]) at
        /project/[...]/src/video_display_pv.cpp:182
What:   started = true;
```

Ainsi, nous avons bien obtenu toutes les informations que nous souhaitions (de façon indirecte pour les noms de variables, mais cela est suffisant). Le principal avantage de cette méthode est de n'avoir requis que très peu de codage supplémentaire ; l'ensemble de cette extension fait en effet moins de 400 lignes.

6.6.3 Arbres des contraintes d'ordonnement

Cette dernière extension permet de construire une représentation graphique de l'arbre des contraintes d'ordonnement, tel que défini à la sous-section 5.5.1. Les arbres de contraintes générés servent essentiellement à illustrer des documents. Le format de sortie choisi a donc été celui d'Xfig avec du \LaTeX pour les éléments textes. Un exemple de graphique obtenu est donné par la figure 6.8, reprise au chapitre suivant (fig. 7.3). L'analyseur a été légèrement modifié pour enregistrer les nouvelles contraintes générées, dans un fichier global à toutes les analyses d'une même validation. Le graphe présenté est obtenu à partir du fichier de contraintes de la figure 6.7.

```

s_1>r_1 q_1>p_1 ←
r_1>p_1 ←
s_1>p_1 ←
←
p_1>q_1 s_1>p_1 ←
r_1>p_1 ←
r_1>q_1 ←
←
r_1>q_1 ←
r_1>p_1 ←
←

```

FIG. 6.7 – Récapitulatif de toutes les contraintes générées lors d'une validation (fichier `tree.ctrs`). Pour simplifier la lecture, les identifiants numériques ont été ici remplacés par les alias correspondants.

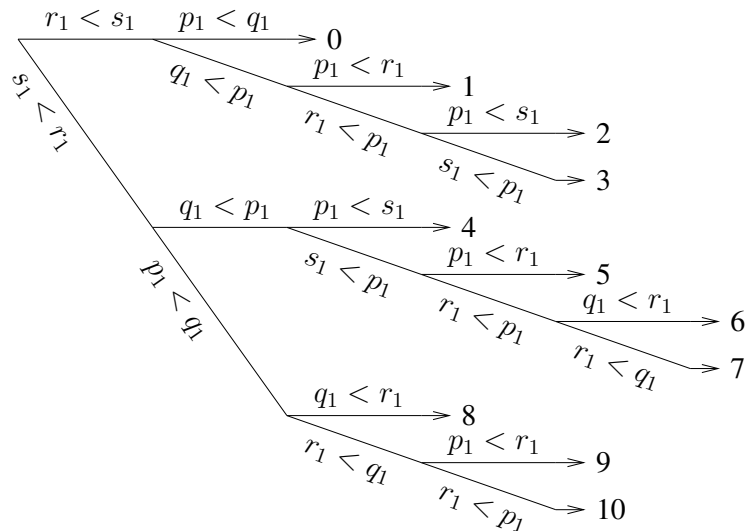


FIG. 6.8 – Arbre de contraintes généré. À noter : la correspondance entre la longueur des traits horizontaux et la longueur des lignes du fichier d'entrée.

Un script Perl, nommé `ctrs2fig`, transforme ensuite ce fichier en un fichier `.fig` représentant l'arbre des contraintes. En simplifiant légèrement, l'algorithme consiste à rajouter un noeud sur la même ligne à chaque contrainte rencontrée, et à passer à la ligne suivante à chaque retour à la ligne rencontré dans le fichier d'entrée (noté \leftarrow dans la figure). Une pile lui permet de savoir à quelle ordonnée raccrocher chaque nouvelle ligne. Quelques calculs de trigonométrie classiques permettent enfin d'orienter et placer les labels correctement.

6.7 Conclusion

Nous avons décrit l'implantation de notre chaîne d'outils. Le composant le plus complexe est l'analyseur car il utilise des algorithmes et des structures de données très spécifiques, mais la princi-

La principale difficulté consiste à faire fonctionner tous les composants ensemble. La chaîne complète compte en effet une petite dizaine d'exécutables : le modèle sous-test compilé avec une implantation SystemC modifiée et complétée, l'instrumenteur `sc2rvs` et l'analyseur `rvsc` codé en C++, plus un ensemble de scripts Shell et Perl et de transformations XSL assurant les liaisons et la génération de sorties humainement compréhensibles. Cela représente environ 3200 lignes de code C++ et 1200 lignes de code divers (perl, xsl, gdb, ...). Le tout est lié au flot de développement TLM, mais est conçu pour rester utilisable pour valider des programmes SystemC classiques.

Au chapitre suivant, nous allons discuter de l'efficacité de la méthode et de sa mise en œuvre.

Chapitre 7

Evaluation et étude de cas

Sommaire

7.1 Les cas élémentaires	104
7.1.1 Exemple avec impasse possible	104
7.1.2 Exemple avec génération d'ordonnements équivalents	105
7.2 Test de performance	107
7.2.1 Indexeur	107
7.2.2 Modèle TLM dédié à des travaux pratiques	109
7.3 Cas réel	110
7.3.1 Description	110
7.3.2 Instrumentation du modèle	111
7.3.3 Validation et dépouillement des résultats	112
7.3.4 Prise en compte des événements persistants	114
7.3.5 Tentative avec la plateforme complète	116
7.4 Bilan	117

Ré-exécuter un test avec un ordonnancement différent peut révéler une erreur dans le modèle. L'ensemble des ordonnancements est fini mais en général beaucoup trop grand pour qu'il soit imaginable de les exécuter tous. L'outil présenté au chapitre précédent extrait un sous-ensemble d'ordonnements dont l'exécution exhaustive garantit toujours de bonnes propriétés de couverture. La principale question soulevée dans ce chapitre est : *le sous-ensemble construit est-il suffisamment plus petit pour qu'il soit possible de simuler effectivement tous ses éléments ?* Plus précisément, il s'agit de déterminer quels critères font que notre outil est utile et applicable.

Nous allons d'abord rapidement vérifier le comportement de notre outil sur quelques exemples de petites tailles mais contenant des synchronisations complexes (section 7.1). Nous essaierons ensuite d'estimer la taille maximale des programmes vérifiables grâce à des exemples dont la taille est paramétrable (section 7.2). Enfin, nous étudierons à la section 7.3 deux modèles fournis par STMicroelectronics. Le premier modèle est de taille moyenne et sert à du décodage vidéo. Son fonctionnement général et ses synchronisations sont encore assez bien compréhensibles. Le second modèle est une plate-forme complète avec tous les composants nécessaires à une *Set-Top Box* (STB, utilisé pour la télévision numérique). Celui-ci a posé de nombreux problèmes aux ingénieurs d'STMicroelectronics, et nous verrons qu'il nous en pose aussi de nombreux.

7.1 Les cas élémentaires

Nous passons sur l'exemple `bozo` (figure 4.4) pour lequel il n'y a rien à dire, mis à part que l'outil fonctionne correctement : il génère les trois ordonnancements voulus.

7.1.1 Exemple avec impasse possible

Le dernier exemple de la section 5.4, dont nous rappelons le code ci-dessous, est plus intéressant car, comme le montre la figure 7.1, le nombre d'ordonnements générés n'est pas nécessairement optimal.

- Processus P : `wait(12); cout <<'p'; if (x) cerr <<"Ko";`
- Processus Q : `wait(e); cout <<'q'; x = 19;`
- Processus R : `wait(12); cout <<'r'; if (!y) e.notify();`
- Processus S : `wait(12); cout <<'s'; y = 1;`

L'exécution de l'outil sur cet exemple donne l'affichage “. . X.”. Chaque caractère affiché correspond à une exécution ; les points désignent les exécutions s'étant normalement déroulées alors que les X désignent des exécutions menant à un blocage causé par les contraintes d'ordonnements héritées. Des informations plus détaillées sont rassemblées dans le répertoire `rvsTraces`. Dans le cas présent, le système sous-test a été exécuté en donnant à l'ordonneur la contrainte $q_2 < p_2$, or le système a atteint un point où seul p_2 était éligible alors que q_2 n'avait pas encore été exécuté.

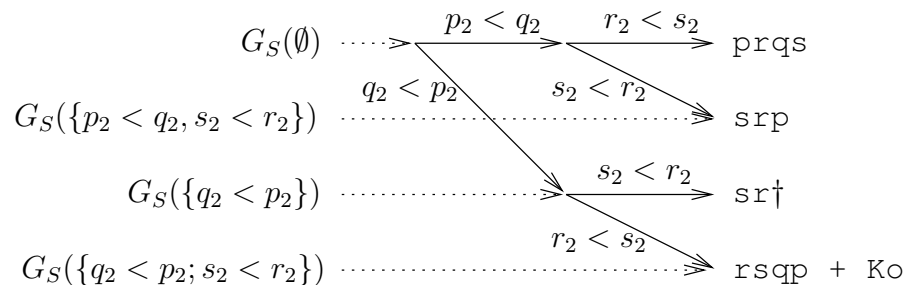


FIG. 7.1 – Arbre de contraintes d'ordonnements avec feuille morte. Les affichages obtenus lors des exécutions sont rattachés aux feuilles.

La chaîne d'outils permet de construire l'arbre des contraintes d'ordonnements, tel que défini à la sous-section 5.5.1. La forme de l'arbre dépend de choix aléatoires effectués par l'ordonneur. C'est-à-dire que deux validations successives d'un même SSTD peuvent donner des arbres différents. En effet, puisque les ordonnancements analysés diffèrent, les contraintes ne sont pas générées dans le même ordre. Dans cet exemple, la contrainte $s_2 < r_2$ (ou son opposée $r_2 < s_2$) influe sur la présence de q_2 dans l'ordonnement obtenu, et donc sur la contrainte $q_2 < p_2$. Par conséquent, les arbres de taille optimale sont obtenus quand la contrainte portant sur s_2 et r_2 est plus proche de la racine que celle portant sur q_2 et p_2 . Un arbre optimal pour cet exemple est donné par la figure 7.2.

Sur 50 validations de ce même exemple avec des germes initiaux différents, nous n'obtenons que 6 fois l'impasse, aussi appelée “*feuille morte*” de l'arbre (apparaissant en 2ème, 3ème ou dernière position). La présence de feuilles mortes constitue un surcoût pour la validation. Cependant, elles sont rarement présentes, et quand elles le sont, elles restent largement minoritaire. Il serait sans doute possible de développer des heuristiques pour les éviter, en étudiant par exemple les influences entre

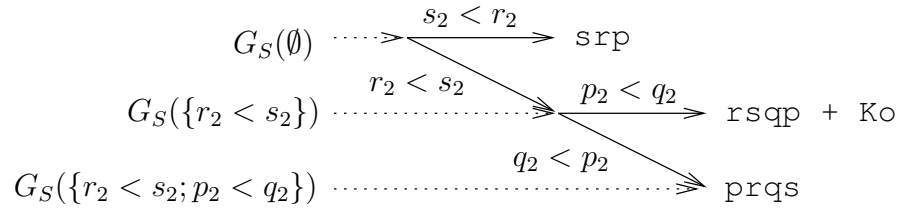


FIG. 7.2 – Arbre de contraintes d'ordonnements sans feuille morte.

transitions ou contraintes, mais cela ne changera de toutes façons pas significativement la taille des modèles vérifiables.

7.1.2 Exemple avec génération d'ordonnements équivalents

Nous allons regarder un nouvel exemple, nommé `contrex`, n'utilisant cette fois que des variables partagées. Celui-ci ne présente pas de possibilité de feuilles mortes (celles-ci sont impossibles en l'absence d'attente sur événements), mais présente un autre type de difficulté.

Les variables sont initialisées par ailleurs à zéro.

- Processus P : `if (!y) {if (a) x=1;}`
- Processus Q : `if (!y) a=1;`
- Processus R : `if (!x) {if (b) y=1;}`
- Processus S : `if (!x) b=1;`

La première remarque est qu'il n'y a dans cet exemple qu'une seule transition par processus. Ensuite, on note que le code exécuté par chaque transition dépend des transitions précédentes, et ceci de façon assez symétrique. La transition p_1 , notée simplement p dans la suite, peut par exemple rendre inaccessible la majeure partie du code de r et s , rendant ainsi ces deux dernières transitions indépendantes. Symétriquement, la transition r peut rendre inaccessible la majeure partie du code des transitions p et q .

Nous lançons de nouveaux 50 validations de cet exemple avec des germes différents pour les tirages aléatoires de l'ordonneur. Nous observons le nombre d'ordonnements générés et simulés. Nous obtenons généralement 10 (43 fois) et plus rarement 11 (7 fois). Comme toutes ces exécutions sont complètes, cela signifie que nous avons généré plus que un ordonnancement par classe d'équivalence, puisque le nombre de classes d'équivalence n'est bien sûr pas aléatoire. Nous allons expliquer, à partir de l'arbre de contraintes représenté par la figure 7.3, ce qui s'est passé.

Le premier ordonnancement (numéro 0), choisi de façon entièrement aléatoire, est $rsqp$. Les transitions r et s ont lu et modifié la variable b , d'où la contrainte $r_1 < s_1$ sur la racine de l'arbre. Considérons maintenant la feuille numéro 1 ; l'ordonnancement correspondant est $qprs$. Il s'agit du seul ordonnancement qui respecte les 3 contraintes associées à ce chemin : $q_1 < p_1$, $p_1 < r_1$ et $r_1 < s_1$. Or, dans cette exécution, p modifie la valeur de x à 1 et en conséquence r et s ne font plus rien d'autre que de lire x ; r et s sont donc indépendantes, et $qprs \equiv qpsr$. La classe d'équivalence composée de ces deux exécutions est divisée en deux dans l'arbre de contraintes. Pour trouver l'autre moitié de la classe d'équivalence, il suffit de rechercher le représentant de $qpsr$ dans l'arbre grâce à la fonction habituelle. Cela mène à la feuille numéro 4. Ainsi, une classe d'équivalence peut être scindée lors d'une validation si, à un moment, l'analyseur hérite d'une contrainte qui n'a plus de sens dans l'exécution courante.

Nous avons expérimentalement montré que le nombre d'exécutions générées varie selon des choix non-contraints faits par l'ordonneur et l'analyseur. Une question théorique intéressante est de sa-

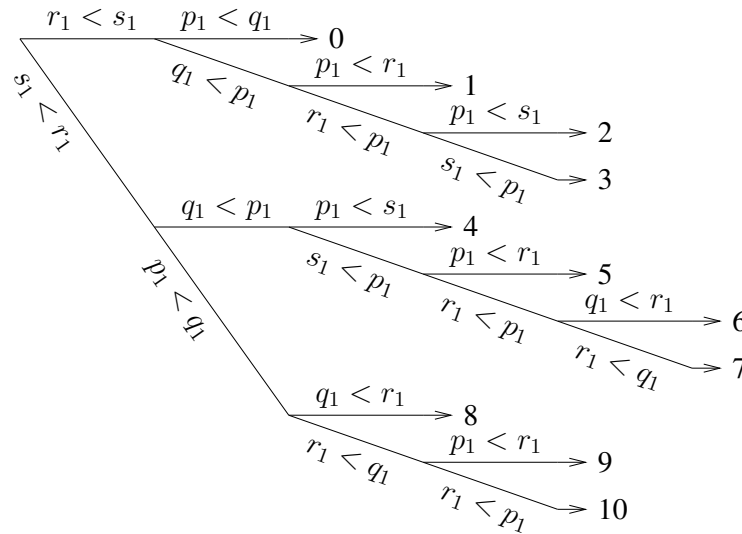


FIG. 7.3 – Arbre de contraintes d’ordonnements avec feuilles équivalentes, pour l’exemple `contrex`. (génééré automatiquement)

voir s’il est possible d’améliorer l’algorithme actuel afin qu’il ne génère que des arbres possédant toujours autant de feuilles que de classes d’équivalence. La réponse est donné par l’exemple ci-dessus puisque’une analyse détaillée de ces classes d’équivalence montrent d’abord qu’elles sont au nombre de 9 (cf tableau 7.1), et enfin qu’elles ne peuvent être rangées sur un arbre de contraintes d’ordonnements sans diviser au moins une classe d’équivalence. Concernant l’arbre donné en exemple, les feuilles numéro 7 et 10 correspondent aussi à une même classe d’équivalence ; les autres classes d’équivalence sont en un seul morceau.

Éléments de la classe d’équivalence	Contraintes correspondantes	N.B.
$pqr\bar{s}, \bar{p}rqs, \bar{p}\bar{r}sq, \bar{r}\bar{p}qs, rpsq, r\bar{s}pq$	$p < q, r < s$	$(p, r) \in \mathcal{J}$
$qp\bar{r}\bar{s}, qps\bar{r}$	$q < p, p < r, p < s$	$(r, s) \in \mathcal{J}$
$pqsr, \bar{p}\bar{s}qr, \bar{s}\bar{p}qr$	$p < q, s < r, q < r$	$(p, s) \in \mathcal{J}$
$psrq, sprq$	$p < r, s < r, r < q$	
$\bar{q}\bar{r}ps, \bar{r}\bar{q}ps$	$q < p, r < p, p < s$	$(q, r) \in \mathcal{J}$
$qrsp, r\bar{q}\bar{s}p, r\bar{s}\bar{q}p$	$r < s, q < p, s < p$	$(q, s) \in \mathcal{J}$
$qspr, sqpr$	$s < p, q < p, p < r$	
$qsrp, sqrp$	$q < r, s < r, r < p$	
$sr\bar{p}\bar{q}, sr\bar{q}\bar{p}$	$s < r, r < p, r < q$	$(p, q) \in \mathcal{J}$

TAB. 7.1 – Description exhaustive des classes d’équivalence de l’exemple `contrex`.

Pour obtenir un arbre avec uniquement neuf feuilles, il faudrait que le couple de transitions (x, y) rattaché au noeud racine soit dépendant pour toutes les exécutions. Sinon, il existerait une classe d’équivalence contenant au moins deux exécutions de la forme $uxyv$ et $uyxv$, et ces deux exécutions seraient nécessairement associées à deux feuilles différentes. Or, dans l’exemple étudié, nous constatons par analyse exhaustive que tout couple de transitions est indépendant dans au moins un cas. Par conséquent, tout arbre de contraintes pour cet exemple contient au moins 10 feuilles et une classe d’équivalence scindée. Plus généralement, tout algorithme dont la correction est basée sur des arbres

de contraintes **purs** ne peut donc être optimal vis-à-vis du nombre d'exécutions par rapport au nombre de classes d'équivalence.

Cela ne signifie pas que des améliorations sont impossibles. Il existe au moins deux idées réalistes :

- détecter lors de chaque analyse si les contraintes héritées portent sur une paire de transitions indépendantes, auquel cas nous savons que l'exécution analysée est équivalente avec une autre exécution générée, antérieure ou postérieure ;
- combiner la technique que nous utilisons déjà avec une autre technique de réduction d'ordre partiel. Le meilleur candidat semble être la technique des *sleep sets* [God96], qui évite une analyse statique préalable des dépendances.

Aucune de ces techniques n'a pour le moment été mise en œuvre, pour la simple raison que les programmes réels, que nous avons étudiés, n'en ont encore jamais montré le besoin.

7.2 Test de performance

Les exemples précédents révèlent certains défauts de la méthode, mais sont peu réalistes. Dans cette section, nous allons essayer d'estimer la taille maximale des programmes vérifiables, grâce à des exemples plus réalistes dont la taille est paramétrable.

7.2.1 Indexeur

Nous avons évalué notre prototype sur l'un des exemples proposés dans le papier fondateur des réductions ordres partiels dynamiques [FG05]. A savoir : l'indexeur. Quelques modifications sont nécessaires pour en faire un programme SystemC ayant un comportement semblable. Notamment, il faut tenir compte de ce que l'ordonnanceur SystemC n'est pas préemptif, contrairement au langage utilisé pour la version originale. La version SystemC est donnée par l'annexe B.

Ce programme se compose de n éléments, comportant chacun deux processus, et d'un tableau global, de taille fixe et servant de table de hachage. Chaque élément génère 4 messages et les enregistre dans la table de hachage globale. En cas de conflit dans la table, c'est-à-dire si deux messages ont le même hachage, le message est enregistré dans la première case vide qui suit. Il n'y a pas de communications entre les éléments en dehors des accès à cette table globale.

Au sein d'un élément, il n'y a qu'un ordonnancement possible pour les deux processus. La présence de 2 processus par élément, contrairement à 1 dans l'exemple original, est motivée par le besoin de rendre la main entre chaque accès à la table de hachage. En effet, utiliser une instruction comme `wait(20, SC_NS)` ou `wait(SC_ZERO_TIME)` pour rendre la main aurait réduit le nombre d'entrelacements possibles par rapport à la version originale. Une alternative, pour rendre la main sans déclencher une synchronisation globale, serait d'utiliser la fonction `yield` que nous avons définie à la section 4.1.3.

Toutes les expérimentations ont été faites sur des stations de travail standards (Linux sur Pentium cadencé à 3 GHz). Les résultats sont donnés par la table 7.2.

La première constatation est qu'une seule exécution suffit tant que le nombre d'éléments n est inférieur ou égal à 11. La raison est simple : jusqu'à cette taille, il n'y a aucun conflit dans la table de hachage. Par conséquent, chaque case mémoire n'est accédée que par un seul processus, et il n'y a donc aucune paire de transitions dépendantes permutables.

A partir de $n = 12$, il y a des conflits dans la table et donc des paires de transitions à permuter. La figure 7.4 représente l'arbre des contraintes pour l'*indexeur* avec 12 éléments. On observe que deux sous-arbres de même taille sont toujours équivalents (ils ne diffèrent que par l'ordre des fils, ce qui n'a

nombre d'éléments	exécutions générées	temps total	conflits
1 ... 11	1	≤ 0.20 sec	0
12	8	0.16 sec	3
13	64	1.06 sec	6
14	512	48.39 sec	9
15	4096	6 min 41 sec	12
16	32768	55 min 44 sec	15

TAB. 7.2 – Résultats pour la validation de l'*indexeur*.

pas de conséquences sur leur sémantique). Cela provient du fait que, pour ce programme, la relation de dépendance est identique pour toutes les classes d'équivalence. Lorsque cette propriété est vraie, un rapide calcul montre qu'il faut générer $|G_1| = 2^x$ exécutions, où x est le nombre de dépendances permutables trouvées dans la première exécution. Ce calcul peut permettre une bonne estimation du coût total de la validation connaissant uniquement le résultat de la première analyse, et rappelle que nous n'échappons pas au problème de l'explosion combinatoire.

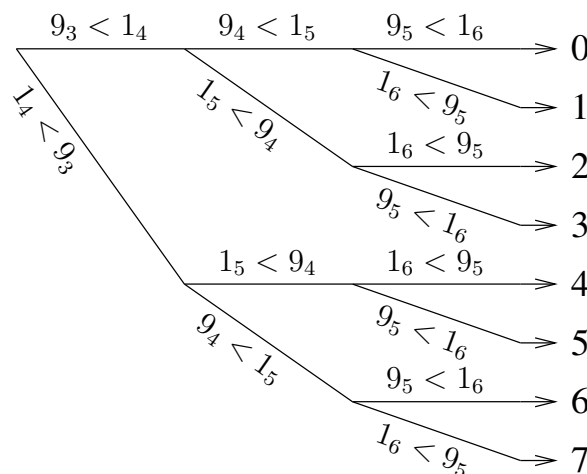


FIG. 7.4 – Arbre de contraintes d'ordonnancements pour l'*indexeur* avec 12 éléments. Les grands chiffres dans les contraintes correspondent au numéro de l'élément.

L'observation de l'état final de la table de hachage montre que l'on n'obtient pas deux fois la même table. Autrement dit, tous les états finaux diffèrent et donc le nombre d'exécutions est exactement égal aux nombres de classes d'équivalences, contrairement aux exemples précédents qui ont été écrits pour. Les différents états finaux diffèrent uniquement par l'ordre des messages dans la table. Par conséquent, toutes les exécutions peuvent être considérées comme correctes d'un point de vue fonctionnel.

Étant donné que jusqu'à présent nous ne disposons pas d'un système de point de sauvegarde permettant de ne pas ré-exécuter chaque exécution depuis l'état initial, les valeurs obtenues ne peuvent pas être directement comparées avec celles de [FG05]. Nous pouvons juste dire que les résultats sont semblables : on observe avec les deux outils un palier jusqu'à $n = 11$ suivi d'une croissance exponentielle.

Sur cet exemple, nous avons pu valider un système avec ses données jusqu'à 16 éléments soit 32 processus. Cependant cela est très dépendant du programme et des constantes qui s'y trouvent, comme par exemple le nombre de messages ou la taille de la table de hachage.

7.2.2 Modèle TLM dédié à des travaux pratiques

Nous allons maintenant étudier un petit modèle TLM, nommé TP-TLM, initialement conçu pour des travaux pratiques d'étudiants en dernière année d'école d'ingénieur. Ce modèle a deux avantages :

- il est relativement proche des exemples industriels visés, bien que très simplifié ;
- il n'est pas soumis au secret industriel, et peut donc être diffusé.

Malheureusement, ce programme n'est pas prévu pour être paramétrable en taille. A défaut de mieux, la solution retenue ici est de dupliquer n fois le système initial, sans ajouter de communications entre les copies (cf figure 7.5).

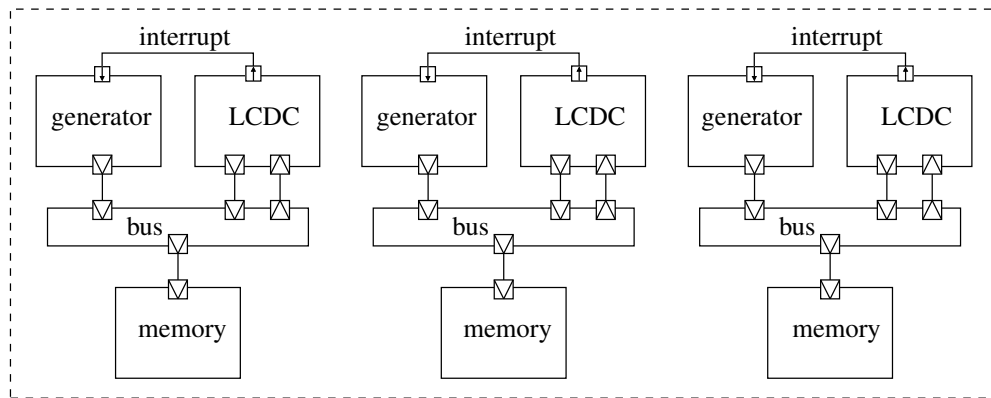


FIG. 7.5 – Architecture du TP-TLM avec $n = 3$.

Ce modèle se compose de 3 composants : un générateur, un contrôleur d'écran (LCDC) et d'une mémoire. Le générateur enregistre des images dans la mémoire et programme le LCDC pour qu'il les affiche. Un `sc_signal<bool>` permet au LCDC de notifier une interruption au générateur dès qu'il a fini l'affichage d'une image. Seule une modification a été nécessaire avant de lancer la validation : l'affichage graphique a été désactivé, afin d'éviter d'ouvrir inutilement une fenêtre à chaque simulation.

nombre d'éléments	exécutions générées	temps total
1	4	0.30 sec
2	16	1.21 sec
3	64	3.89 sec
4	256	16.09 sec
5	1024	1 min 12 sec
6	4096	4 min 59 sec

TAB. 7.3 – Résultats pour la validation du TP-TLM.

Les résultats de la validation sont donnés par le tableau 7.3. Pour $n = 1$, c'est-à-dire avec juste un exemplaire du système, 4 exécutions ont été générées. Les outils annexes montrent qu'ils proviennent de deux paires de transitions dépendantes permutable, l'une dans le générateur, l'autre dans le LCDC.

La première paire correspond aux deux actions ci-dessous, appartenant à deux transitions différentes (extraits des traces détaillées obtenu avec `rvi`) :

```
Where: in Generator::interrupt_handler (...) at Generator.cpp:74
What: interrupt = true;
```

```
Where: in Generator::compute (...) at Generator.cpp:33
What:  if (!interrupt)
```

La deuxième correspond aux deux actions ci-dessous :

```
Where: in LCDC::write (...) at LCDC.cpp:171
What:  started = true;
```

```
Where: in LCDC::compute (...) at LCDC.cpp:191
What:  while (!started)
```

Nous observons les deux mêmes dépendances permutables dans toutes les exécutions. Le nombre d'exécutions générées est donc logiquement $|G| = 2^2 = 4$, selon la formule donnée à la section précédente. Dans ces deux cas de dépendance, le code est écrit de tel sorte qu'il fonctionne avec les deux ordonnancements possibles. Les flèches rouges générées peuvent donc être qualifiées de "fausses alertes". Chacune de ses fausses alertes double le nombre d'exécution générées.

Les résultats pour $n > 1$ étaient prévisibles. En effet, soit \mathcal{E}_n l'ensemble des classes d'équivalence pour le système composé de n fois le programme original, on a : $\mathcal{E}_n = \mathcal{E}_1^n$ et donc $|\mathcal{E}_n| = |\mathcal{E}_1|^n = 4^n$.

Bien que cet exemple soit très basique, nous pouvons en retirer deux conclusions :

- les fausses alertes coûtent cher ;
- les réductions d'ordre partiel ne dispensent pas d'isoler les diverses parties indépendantes d'un gros système.

7.3 Cas réel

Nous allons maintenant parler de notre principale étude de cas. Il s'agit d'un programme fourni par STMicroelectronics, et qui modélise une partie du flux vidéo d'une *Set-Top Box*, c'est-à-dire le système sur puce d'une télévision haute définition. Elle est nommée LCMPEG, du nom de son principal composant : *Low Cost MPEG decoder*. Il s'agit d'un modèle fonctionnel mais utilisant tout de même du temps pour son fonctionnement. Il est aussi utilisé comme étude de cas pour les travaux sur l'élaboration de la méthodologie PV/PVT (cf sous-section 2.2.3).

7.3.1 Description

La plateforme se compose des 5 composants représentés sur la figure 7.6 : un générateur modélisant un processeur, le décodeur LCMPEG, un contrôleur d'écran, une mémoire et une "stop box". Ce dernier composant sert juste à terminer proprement la simulation et ne correspond à rien de la puce finale. Le composant mémoire est classique. Les trois autres sont en revanche spécifiques à ce modèle et sont donc ceux que nous souhaitons valider.

Comme dans tous les modèles fonctionnels récents développés à STMicroelectronics, le bus est "physiquement idéal", c'est-à-dire que toutes les transactions sont acheminées sans délai, et ceci quels que soient leur nombre ou leur taille. Il en est de même de la mémoire, si bien qu'une série d'écritures ou lectures en mémoire peut être faite de façon atomique (autrement dit : en une seule transition regroupant plusieurs transactions). Une autre conséquence de son niveau d'abstraction est que l'on trouve de longues sections de code séquentiel (principalement dans le décodeur) et peu de synchronisations.

Le comportement est périodique. Il s'agit de décoder des images puis de les afficher. Le générateur programme le décodeur LCMPEG avant chaque image, attend que celui-ci termine le décodage, puis

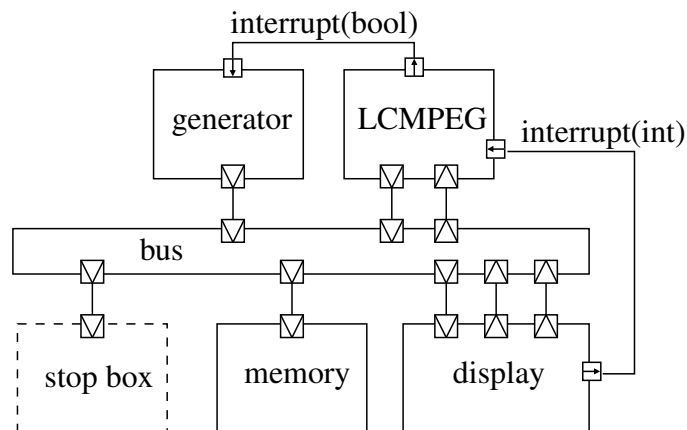


FIG. 7.6 – Architecture du modèle LCMPEG.

que le contrôleur d'écran finisse l'affichage de l'image décodée. Une fois cela fait, le générateur réitère ces actions avec l'image suivante. Le LCMPEG décode l'image qu'il lit puis écrit dans la mémoire. A la fin de chaque ligne de blocs d'images, le LCMPEG se synchronise avec le contrôleur d'écran afin que celui-ci ne soit pas obligé d'attendre la fin du décodage de l'image pour commencer à la charger dans sa mémoire vidéo.

Un fil d'interruption permet au contrôleur d'écran et au LCMPEG de se communiquer le numéro de ligne courant. De même, un fil d'interruption entre le LCMPEG et le générateur permet au LCMPEG de signaler au générateur la fin du décodage d'une image. En revanche, il n'existe aucun fil d'interruption entre le contrôleur d'écran et le générateur. Par conséquent, le générateur est obligé d'utiliser une technique de *polling*¹ pour attendre la fin de l'affichage d'une image, c'est-à-dire qu'il utilise un code de la forme ci-dessous :

```
// L'information est stockée dans un registre du contrôleur mais il n'y a pas de sc_event associé.
value = port.read(addr_display+register_offset);
while (!value) {
    wait(T, SC_NS); // Le choix de la valeur de T est souvent délicat.
    value = port.read(addr_display+register_offset);
}
```

Par défaut, le résultat du décodage est affiché dans une fenêtre X11. Pour des simulations automatiques, un mode permet de supprimer cet affichage. Les images décodées sont enregistrées dans un fichier, ce qui permet de les comparer automatiquement aux résultats de références. Cela constituera notre oracle pour savoir si une exécution est fonctionnellement correcte.

7.3.2 Instrumentation du modèle

La première étape consiste à instrumenter le modèle pour la détection des accès aux variables partagées. L'outil et les techniques d'instrumentation ont été décrits à la sous-section 6.3.3.

L'exécution de l'outil `sc2rvs` (cf figure 6.4) se déroule sans souci particulier. Le résultat se

¹La traduction officielle est "*scrutation*", mais en pratique les francophones utilisent majoritairement le terme anglais. L'expression "*attente active*" est parfois aussi utilisée.

compose d'une liste de variables partagées et d'événements, et d'une liste d'instructions à ajouter au code source.

En revanche, l'intégration de ces instructions au code original via l'outil `patch_tool.pl` n'est pas satisfaisant, et ce malgré les améliorations apportées. En effet, certaines portions du code deviennent syntaxiquement incorrectes, ou pire : obtiennent une sémantique différente. Les plus gros problèmes sont rencontrés dans le composant `tac_memory`, car une partie conséquente du code se situe dans des macros, et l'outil d'instrumentation travaille sur le code prétraité par le préprocesseur. En l'état, l'utilisation de cet outil oblige l'utilisateur à relire l'ensemble du code source, afin d'apporter les corrections nécessaires. Cela peut représenter plusieurs heures de travail.

Finalement, l'instrumentation a été réalisée grâce aux sondes `probe<T>` présentées au chapitre précédent. Connaissant la liste des variables partagées, il suffit de modifier manuellement leur déclaration, par exemple en remplaçant `int started;` par `probe<int> started;`. Cela ne demande pas plus de temps que les corrections liées à la méthode précédente, car les lignes de code impactées sont ici beaucoup moins nombreuses (18 au lieu de 300 dans cet étude de cas, hors composant mémoire). De plus, le code instrumenté est beaucoup plus facile à maintenir. L'outil d'instrumentation ne doit pas pour autant être considéré comme obsolète : d'une part il est nécessaire pour repérer rapidement quelles sont les variables partagées ; d'autre part il pourrait être modifié pour instrumenter automatiquement les déclarations plutôt que les accès.

Les dernières versions de nos outils permettent une instrumentation compositionnelle. Par conséquent, si un composant est réutilisé dans une nouvelle plateforme, aucune modification n'est à faire sur sa version instrumentée. C'est notamment intéressant pour le composant mémoire qui doit être instrumenté manuellement afin d'utiliser les outils spécifiques à l'enregistrement des accès mémoires (cf section 6.3.3.4).

Nous avons du aussi apporter une légère modification à la librairie TLM chargée de la lecture des options de la ligne de commande. En effet, celle-ci entrainait en conflit avec celle utilisée dans notre noyau SystemC modifié. Nous avons remplacé la levée d'une erreur bloquante par un simple message d'avertissement signalant la présence d'une option inconnue.

7.3.3 Validation et dépouillement des résultats

Le test normal contient un long flux vidéo conçu pour vérifier le bon comportement de l'algorithme de décodage. Comme nous recherchons plutôt les erreurs de synchronisation, nous nous limitons pour le moment au décodage des 3 premières images, ce qui représente environ 150 transitions et 70 000 transactions. La durée d'une simulation de ce type est estimée à **0.39 secondes**. La validation est lancée avec la commande :

```
rvs -sut tlm_run.exe -check "diff -r ram_images ref_images"
```

Notre outil (`rvs`) génère **128 exécutions**, ce qui lui prend **1 minute et 8 secondes**.

Une question intéressante est de savoir comment se répartit le temps total T entre les simulations S , et le reste R (dont enregistrement de la trace, analyse et génération des nouveaux ordonnancements). Nous savons que $S = 128 \times 0.39 = 50sec$, et donc $R = T - S = 1min08sec - 50sec = 18sec$. Pour ce modèle, nous avons donc environ 73,5% pour les simulations et 26,5% pour le reste, ce qui est très acceptable, surtout si l'on considère le nombre réduit d'ordonnancements simulés : 128 parmi les plus de 2^{40} possibles. Ces ratios sont bien sûr très dépendants du modèle étudié, puisqu'ils dépendent directement de la durée des simulations.

A la fin de chaque exécution, nous comparons les images décodées aux images de référence. Grâce à cela, nous savons que ces 128 exécutions sont fonctionnellement correctes (mais nous verrons par la suite que cet oracle n'est pas suffisant). Cependant, une analyse plus détaillée révèle des différences entre exécutions générées. A la fin de chaque simulation, le modèle affiche quelques statistiques, telles que ci-dessous :

```
Simulation Time:      0.000000460 s
Real simulation time:  0.9200 s
Transactions:        71800
Transactions/Sec:    78043.48
Kbytes:              4486.72
Kbytes/Sec:          4876.87
Byte/Transaction (Av.): 63.99
```

La ligne *Simulation Time* correspond au temps simulé par SystemC, alors que la ligne *Real simulation time* correspond au temps réel utilisé par le simulateur (ici mesuré sur un ordinateur portable peu rapide). Une comparaison de ces chiffres montrent que le temps simulé varie entre 0.000000450 s (soit 450ns) et 0.000000480 s (soit 480ns)². Le nombre de transactions varie entre 71742 et 71804, de façon discontinue.

Considérons l'exécution avec le plus de transactions. La trace analysée contient la ligne suivante :

```
<red first="150" second="151" item="var316"/>
```

Cela nous informe qu'il y a une dépendance entre la transition 150 et la transition 151, et que cette dépendance est causée par la variable d'identifiant 316.

La seule façon de savoir qui se cache derrière l'identifiant 316 est de générer puis consulter la trace détaillée générée avec gdb (cf sous-section 6.6.2). Pour la transition 150, nous obtenons :

```
Step 150 begins here (process 0):
```

```
generator_exp_lcmpeg_pv::waitForVideoDisplay (...
in generator_exp_lcmpeg_pv::startLCMPEG (...
in generator_exp_lcmpeg_pv::Compute (...
```

... and executes:

```
Action: read (variable 316)
Where:  in video_display_pv::read (...
What:   data = started ? 0x00000001 : 0x00000000;
```

```
Action: wait_time
Where:  in generator_exp_lcmpeg_pv::waitForVideoDisplay (...
What:   wait(10, SC_NS);
```

Et pour la trace 151 (extrait) :

```
Action: write (variable 316)
Where:  in video_display_pv::Compute (...
What:   started = false;
```

²Ces valeurs ne sont pas physiquement réalistes, mais ce n'est pas le but de ce modèle qui se veut purement fonctionnel.

Comme indiqué, la transition 150 commence dans le générateur, et plus précisément dans la fonction qui attend la fin de l’affichage d’une image. La première communication inter-processus enregistrée se situe dans la fonction `read` du contrôleur d’écran. Cela signifie que le processus du générateur a fait une transaction pour lire un registre du contrôleur d’écran. La transition se termine par une attente de $10ns$, de nouveau dans le générateur et donc après la fin de la transaction.

Lors de la transition 151, la variable `started` est affectée à `false`, ce qui signifie que l’affichage est terminé. Comme la lecture a eu lieu avant l’écriture avec cet ordonnancement, le générateur n’a pu voir que l’affichage était fini. Le générateur faisant du *polling*, il lira de nouveau la variable, via une nouvelle transaction, $10ns$ plus tard. Cela n’a pas de conséquence fonctionnelle, mais explique les $30ns$ d’écart entre l’exécution la plus rapide et la plus lente, puisque ce retard de $10ns$ est possible à la fin de l’affichage de chacune des 3 images.

Cependant, le problème du *polling* entre le générateur et le contrôleur d’écran n’explique pas toutes les variations sur le nombre des transactions : nous devrions obtenir un diamètre de 6 (car il s’avère que chaque transaction est compté deux fois) mais nous observons un diamètre de 62. Cela nous a poussé à développer un nouvel oracle de façon plus rigoureuse. Le flux vidéo fourni contient plusieurs codages différents d’une même image, le but étant de faciliter le contrôle du résultat : l’utilisateur doit voir une image fixe. Malheureusement, cela a un effet pervers : le contrôleur d’écran peut afficher une ancienne image décodée au lieu de la nouvelle sans qu’il y ait de différences observables.

Nous avons résolu le problème en modifiant volontairement l’image décodée en fonction du numéro d’image, au moment où le décodeur enregistre l’image décodée dans la mémoire. Cette modification n’a aucune conséquence sur les synchronisations, mais permet de savoir quel image l’écran est en train d’afficher. Une nouvelle validation de ce modèle montre que la troisième image peut être affichée à la place de la deuxième, pour certains ordonnancements valides. Autrement dit, le décodeur LCMPEG va trop vite par rapport au contrôleur d’écran. Il s’agit bien d’un bug dans le modèle analysé. La lecture des traces détaillées montre que l’erreur provient d’une permutation possible entre la remise à zéro du signal d’interruption issu du contrôleur, et la lecture de ce signal par le décodeur. Cette erreur explique aussi indirectement l’écart entre les nombres totaux de transactions.

7.3.4 Prise en compte des événements persistants

Parmi les 128 exécutions générées, nous en avons trouvées qui diffèrent, puis nous les avons étudiées pour trouver des bugs. Nous en avons aussi trouvées qui ne diffèrent pas, et nous allons maintenant les étudier afin de voir comment les éviter.

Une rapide analyse des traces d’exécutions montrent que les exécutions redondantes observées proviennent de l’utilisation d’un couple événement - variable pour simuler un événement persistant, selon le code donné par la figure 4.7 à la fin de la section sur les blocages au démarrage. L’outil d’analyse y voit une *data-race*, bien que le code soit écrit pour être correct avec les deux ordonnancements possibles. L’objectif est donc de compléter notre analyseur pour qu’il ne force pas la permutation dans ce cas là.

7.3.4.1 Nouvelle classe `pevent`

La première étape consiste à définir une nouvelle classe `pevent`. Sa définition est fournie par la figure 7.7. Les motivations pour définir une nouvelle classe sont multiples :

- la modélisation d’un événement persistant est ainsi toujours codée de la même façon ;
- la lisibilité du code est améliorée ;

- l'instrumentation pour l'enregistrement des traces d'exécutions peut se faire une fois pour toute dans la définition de la classe.

```

1  struct pevent {
2      pevent() : var(false) {}
3      void wait() {
4          if (!var)
5              ::wait(ev);
6          else
7              yield();
8          var = false;
9      }
10     void notify() {
11         var = true;
12         ev.notify();
13     }
14     protected:
15         bool var;
16         sc_event ev;
17 };

```

FIG. 7.7 – Définition de la classe `pevent`

Par rapport au code déjà présent dans le programme, nous avons ajouté une instruction `yield` (ligne 7) dans le cas où la notification a lieu avant l'attente. L'objectif est que le nombre de transitions soit constant par rapport à l'ordonnancement, parce que dans le cadre des ordres partiels, deux ordonnancements ne peuvent être considérés comme équivalents que si leur longueur est égale. Cela modifie la sémantique du programme mais il s'agit d'une modification *conservatrice*, c'est-à-dire que nous pouvons ajouter des *fausses erreurs* (*false positive* en anglais), mais nous ne cachons aucune vraie erreur. De plus, les fausses erreurs sont peu probables avec cette modification, car elle ne fait que rendre la main à un endroit où le programmeur doit de toute façon s'y attendre (puisqu'il ne sait a priori pas si le `wait(ev)` sera exécuté).

7.3.4.2 Intégration dans l'outil d'analyse

La deuxième étape consiste à compléter notre outils d'analyse pour qu'il tienne compte de ces nouveaux événements persistants. Il y a trois actions à considérer : la *notification* (lignes 11 et 12 de la classe `pevent`), l'*attente* (lignes 4-7) et la *reprise* (ou *reset*, ligne 8). Distinguer l'attente et la reprise est nécessaire car l'ensemble n'est pas atomique : un appel à la fonction `wait` est à cheval sur deux transitions.

Les cas de non-commutativité sont énumérés ci-dessous. Le cas 2 ne peut causer une dépendance permutable seulement si deux processus attendent sur un même événement persistant, ce qui serait une très mauvaise idée de la part du programmeur.

1. une reprise suivie d'une notification, ou l'inverse, car la notification est ignorée seulement dans le deuxième cas ;
2. une attente suivie d'une reprise, ou l'inverse, car la reprise peut effacer la notification avant que l'attente la voit ;
3. deux notifications sont dépendantes à cause des conséquences sur l'ordre causal, comme pour les événements normaux (cf section 5.4.1.2).

De plus, la transition qui suit l'attente (la *reprise*) est causalement après la notification correspondante.

Une fois ces informations connues, il est assez aisé de compléter l'implantation, puisque les classes pour le traitement des vecteurs de communication sont bien séparées du cœur de l'analyseur (cf figure 6.5).

7.3.4.3 Résultat

Modifier le programme testé pour qu'il utilise cette nouvelle classe a été facile. Les sections de code à remplacer sont repérées par l'analyseur, puisqu'elles correspondent aux (fausses) *data-races*. Nous avons effectué un remplacement pour deux événements persistants.

Nous avons relancé une analyse complète du modèle. Cette fois ci, notre outil génère **32 exécutions**, et cela lui prend **13 secondes**. Ce nouveau jeu d'ordonnements fourni la même couverture que le précédent. Ce résultat est donc très encourageant.

7.3.5 Tentative avec la plateforme complète

Étant donné le succès de notre chaîne d'outils sur le modèle du LCMPEG, considéré comme étant de taille moyenne, nous avons décidé de confronter notre prototype au système complet dont le LCMPEG est issu. Le système complet est appelé 7100 et se destine au traitement des flux multimédias pour une télévision haute définition.

Celui-ci est beaucoup plus gros puisqu'il compte :

- 58 processus (*SC_THREAD* et *SC_METHOD*);
- 3 composants représentant des processeurs, pouvant chacun exécuter du logiciel embarqué ;
- 67 événements SystemC et 1380 variables partagées ;
- 250 000 lignes de code.

Mais il est surtout très différent de part le niveau d'abstraction utilisé. En effet, il utilise une ancienne version du système de communication TLM et du protocole TAC, qui ne permet notamment pas des séquences atomiques de transactions. Par ailleurs, les algorithmes et les communications sont définies de façon plus fines, par exemple au niveau pixel au lieu du niveau bloc d'image. Une conséquence directe est que pour simuler une même fonctionnalité, par exemple le décodage d'une image, le nombre de transitions nécessaires est considérablement augmenté. En contrepartie, les synchronisations globales, c'est-à-dire les changements de cycle, sont beaucoup plus fréquents. Cela a pour conséquence de réduire le nombre d'ordonnements possibles. Le développement d'une nouvelle version, au même niveau d'abstraction que le modèle du LCMPEG précédemment étudié, est en cours, mais les premiers livrables n'ont pas été disponibles à temps pour notre étude de cas.

7.3.5.1 Instrumentation

L'exécution de l'outil de *sc2rvs* a posé plusieurs problèmes.

- Le modèle ne compilait qu'avec la version 3.2.x de *gcc*, alors que Pinapa utilise l'analyseur syntaxique et sémantique de *gcc* 3.4.1. Il a donc fallu corriger le modèle afin qu'il compile avec la version 3.4.1 de *gcc*.
- L'outil Pinapa ne permet pas la compilation séparée, c'est-à-dire que tous les fichiers **.cpp* doivent être inclus dans un seul et grand fichier. Or, ce modèle n'accepte pas de compiler de cette façon, à cause notamment de plusieurs conflits de noms. La solution retenue fut de séparer le modèle en plusieurs sous-modèles, en remplaçant les portions absentes par des bouchons inodores pour l'outil d'instrumentation.
- Il s'est avéré que certaines portions de l'arbre abstrait étaient ignorées lors de l'instrumentation. Cela était dû à des bugs dans Pinapa et *sc2rvs* qui ont été corrigés depuis.

Ensuite, il n'a pas été possible d'intégrer les lignes d'instrumentation au code source de façon automatique. Comme pour le modèle du LCMPEG, nous avons manuellement instrumenté les déclarations des variables partagées pour utiliser la technique des sondes *probe<T>*, en se basant sur les résultats de *sc2rvs* pour connaître la liste de ces variables.

L'ensemble de l'instrumentation a demandé plusieurs jours de travail. Certaines erreurs dans l'instrumentation n'ont été découvertes qu'au moment de l'analyse des premières traces d'exécutions. Cependant, avec l'outil `sc2rvs` corrigé et la bonne méthodologie connue, nous espérons pouvoir instrumenter une plateforme de cette taille en une seule journée.

7.3.5.2 Exécution et analyse

Avec le logiciel embarqué utilisé pour cette étude de cas, le comportement de ce modèle TLM consiste en une phase assez complexe d'initialisation, suivie d'une phase de décodage et affichage d'un flux vidéo. Il est nécessaire de réduire la longueur des simulations afin de pouvoir espérer les analyser. Nous avons choisi de demander au simulateur d'arrêter la simulation au bout d'une demi seconde de temps simulé car cela est juste suffisant pour terminer la première phase et entamer la deuxième.

L'analyse des premières traces d'exécutions apportent des points positifs et des points négatifs. Tout d'abord, les traces d'exécutions enregistrées sont très lourdes : environ 220 Mo. La première générée contient 671031 transitions, réparties sur 649203 cycles. En conséquence, la très grande majorité des transitions (près de 97%) sont seules dans leur cycle, et ne sont permutable avec personne d'autre. Autrement dit, ce modèle s'avère être quasiment synchrone.

Notre analyseur perd beaucoup de temps à analyser de très longues traces alors que de petites portions seulement sont pertinentes pour les éventuelles dépendances à l'ordonnancement. Cependant, il parvient tout de même à analyser la trace et découvre (seulement) **14 dépendances permutable**. Ceci est une bonne nouvelle puisqu'elle laisse espérer une validation complète de ce test (c'est-à-dire ce modèle avec ce logiciel embarqué et ces données fixées) en environ 16000 exécutions. Cette validation n'a pas été faite pour plusieurs raisons :

- à raison d'une minute par couple simulation-analyse (30 sec pour chaque), il faudrait prévoir 12 jours de simulation avec l'outil actuel ;
- il faudrait disposer d'un oracle fiable pour que l'effort soit rentable, or pour le moment nous ne savons pas définir une exécution fonctionnellement correcte pour ce modèle ; étant donné la taille des traces, il est indispensable de les effacer dès la fin de leur analyse et en conséquence aucune vérification ne peut être faite après coup ;
- l'outil d'enregistrement de traces détaillés échoue, principalement à cause de la très grande taille des traces. Par conséquent, il est très difficile d'expliquer les dépendances permutable que nous détectons.

En résumé, nous ne sommes pas parvenu à valider ce modèle, mais les plus grosses difficultés viennent du fait que ce modèle n'est pas au niveau d'abstraction prévu pour notre outil. La nouvelle version (STi7200 [STM07], figure 7.8), si elle est codée à un niveau d'abstraction semblable au modèle du LCMPEG, pourrait amener à des résultats très différents.

7.4 Bilan

Nous avons réussi la validation de programme réel de taille moyenne et espérons pouvoir valider dans l'avenir des programmes de plus grande taille. En attendant, une question intéressante est de savoir ce qui détermine la difficulté de valider un modèle. Le nombre de lignes de code n'est pas vraiment significatif, car la présence de longues sections de code séquentiel n'a que peu d'influence sur notre prototype. Le nombre de processus ou la longueur des exécutions exprimée en nombre de transitions sont déjà plus représentatif. Actuellement, nous supportons jusqu'à environ 15 processus



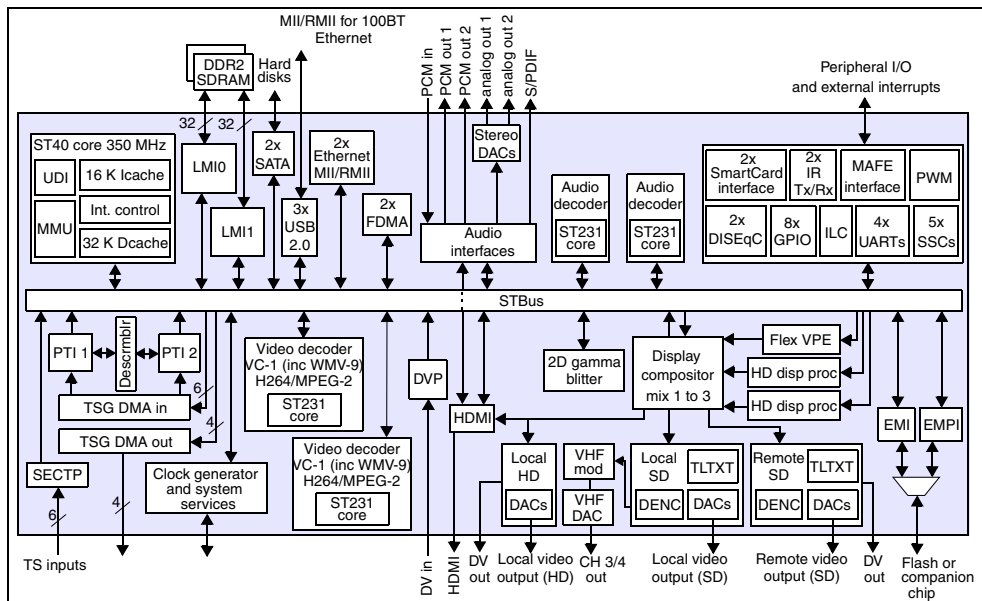
STi7200

Triple display, HDTV set-top box, dual decoder for H.264 and VC-1

Data Brief

Features

- Single-chip, high-definition STB decoder:
 - H.264 and Microsoft® VC-1 compatible
 - Linux®, Windows® CE and OS21 compatible 350 MHz ST40 CPU core
 - supports NAND flash, NOR flash and Sflash
 - local memory 2 x DDR2 333 MHz
 - transport filtering and descrambling
 - dual H.264, MPEG-2, VC-1 video decoding
 - SVP compliant
 - Windows® DRM support
 - triple display composition
 - integrated VHF channel 3/4 modulator
 - dual audio decoder, including Windows Media® Audio 9 (WMA-9) and WMA-9 Pro
 - DVD data retrieval and decryption
- HD DVD/BD compliant
- DVR capable
- HDMI/HDCP interface with CEC line controller
- IQI (Image Quality Improvement) support
- Connectivity:
 - triple USB 2.0 host controller/PHY interface
 - digital audio and video auxiliary inputs
 - low-cost modem support
 - dual 100BT ethernet controller, MAC and MII/RMII interface for external PHY
 - dual serial ATA (SATA)
 - high speed synchronous interface (MPX) to STVi498 cable and DOCSIS front-end chip



December 2006

CD00145658 Rev 1

1/8

For further information contact your local STMicroelectronics sales office.

www.st.com

FIG. 7.8 – Extrait de la présentation publique de la puce STi7200

tel-00350929, version 1 - 7 Jan 2009

et 150 transitions, mais ce ne sont que des ordres de grandeurs. La fréquence des synchronisations globales ainsi que la forme des communications inter-processus peuvent ensuite faire la différence. Il s'avère que les techniques présentées sont le plus utiles avec les modèles les plus fonctionnels, c'est-à-dire les plus abstraits et surtout les plus asynchrones.

Le principal point faible actuel de notre flot de validation est l'outil d'instrumentation. Comme nous l'avons vu, un travail manuel est nécessaire. Cela pose un problème évident qui est le surplus de travail pour les utilisateurs, mais il y a un second problème. En effet, nous ne pouvons garantir qu'un travail manuel est correct, surtout s'il n'est pas fait par des spécialistes. L'oubli d'une variable partagée lors de l'instrumentation peut violer la propriété de couverture complète garantie par les algorithmes utilisés, sans que cela soit détectable.

Par ailleurs, l'étude de cas du LCMPEG a aussi montrée qu'il fallait être rigoureux dans la définition et implantation de l'oracle³. Nous n'avons en effet découvert que très tardivement l'erreur présente entre le décodeur et le contrôleur d'écran, menant au saut d'une image (dans ce cas précis, la cause de l'erreur a été trouvée, via l'étude des traces détaillées, avant l'erreur elle-même). Les travaux d'un autre doctorant de l'équipe SPG de STMicroelectronics, à savoir Younes LAHBIB, portant sur la définition de propriétés au niveau TLM, devraient aider à cette tâche [DGG⁺05].

Enfin, nous avons vu que les outils annexes permettent de remonter à la source des problèmes. Cela est très important car détecter la présence d'un bug n'est pas suffisant ; il faut aussi permettre sa correction. Ces outils peuvent aussi avoir un intérêt pédagogique et peut-être que grâce à eux, les développeurs TLM vont accroître leur productivité.

³le problème d'évaluer la qualité de l'oracle est connu dans la littérature sous le nom de *vacuité* (*vacuity* en anglais). Voir par exemple [Kup06]

Chapitre 8

Génération de schémas de temporisation

Sommaire

8.1	Contexte et motivations	122
8.1.1	Système plongé dans un environnement non-déterministe	122
8.1.2	Système partiellement temporisé	123
8.1.3	Exemple introductif	124
8.1.4	Méthodes existantes pour le choix des délais effectifs	125
8.2	Principe	126
8.2.1	Séparation de l'ordonnancement, des durées et des données	126
8.2.2	Les modèles TLM que nous considérons	126
8.2.3	Aperçu général	128
8.2.4	Algorithme formel	130
8.3	Implantation	134
8.3.1	Construction des contraintes linéaires	134
8.3.2	Résolution des systèmes linéaires	135
8.3.3	Représentation des schémas de temporisation	135
8.4	Évaluation	136
8.5	Autres approches pour la validation de modèles avec temps imprécis	136
8.5.1	Test, plus réduction d'ordre partiel ou programmation linéaire	136
8.5.2	Extraction d'un modèle puis vérification formelle	137

Nous avons présenté dans les chapitres précédents une méthode pour la génération d'ordonnements. Les ordonnements peuvent être vus comme un type particulier de "données" qu'il faut fournir au système sous test pour l'exécuter. De façon très abstraite, la méthode consiste à générer des tests de proche en proche, en analysant le comportement dynamique de chacun. La question est maintenant de savoir si cette méthode peut être utilisée pour la génération d'autres types de données.

Nous présentons dans ce chapitre une extension de notre méthode de génération des ordonnements pour la génération de valeurs temporelles, c'est-à-dire de *durées*. Dans le cadre de la modélisation au niveau transactionnel, le développeur souhaite parfois modéliser le fait qu'un processus attend sur du temps, mais sans connaître précisément ce temps. Une solution consiste à choisir une valeur proche de ce qu'elle devrait être dans la réalité, en la faisant éventuellement varier aléatoirement dans l'intervalle des valeurs réalistes. Cela n'est guère efficace et ne fournit aucune garantie sur le fait

d’avoir ou non trouvé tous les cas possibles. Notre objectif est de générer, en plus des ordonnancements, des jeux de valeurs temporelles pertinents, c’est-à-dire couvrant le plus grand nombre possible d’exécutions sensiblement différentes.

Ce chapitre se découpe en quatre sections. La première section motive le développement d’un outil pour la génération de valeurs temporelles en décrivant les deux principaux cas où cela est utile, et en indiquant les faiblesses des méthodes actuelles. La deuxième section présente et explique notre méthode d’un point de vue théorique. La section suivante détaille l’implantation du prototype réalisé, puis la section 8.4 fournit les résultats expérimentaux obtenus. Le chapitre se termine par une présentation des autres approches existantes pour le même problème.

8.1 Contexte et motivations

Les modèles TLM avec délais variables se rencontrent dans au moins deux cas : les systèmes partiellement temporisés et les systèmes plongés dans un environnement asynchrone.

8.1.1 Système plongé dans un environnement non-déterministe

Tous les systèmes sur puces communiquent avec l’extérieur. L’extérieur est un terme générique qui peut prendre diverses formes. Il peut s’agir d’un autre système informatique, d’un être humain ou d’un phénomène naturel. Dans tous les cas, les informations arrivent au système sur puce via des capteurs qui se chargent de coder les entrées sous une forme numérique. Suivant les cas, plus ou moins d’informations sont disponibles sur les durées qui s’écoulent entre deux entrées successives. Dans certain cas, c’est le système sur puce qui décide d’effectuer une mesure. Dans d’autre cas, le système doit attendre qu’un autre système extérieur lui envoie des données. Si ce système externe est aussi un système informatique, le protocole de communication peut éventuellement fixer des délais minimaux ou maximaux entre deux réceptions de données. Si le système externe est un être humain, il n’y a généralement aucune borne sur les délais.

Lors de la réalisation de modèles TLM, il faut tenir compte des libertés existantes sur les instants d’arrivée des entrées. Étant donné un test fixant la valeur des entrées, faire varier les durées entre entrées successives peut révéler des erreurs. En réduisant les durées, une saturation peut apparaître dans le modèle. Cela peut aussi faire passer un processus avant un autre, et causer une erreur si cela n’avait pas été prévu. En augmentant les durées, un “time-out” peut être dépassé, ce qui peut provoquer l’exécution d’une autre portion de logiciel embarqué.

Afin de vérifier que les modèles développés sont robustes à ce type de variation, il est important de faire varier les durées lors de la génération de test. Dans le cas de génération dynamique de tests, l’extérieur est modélisé par un ou plusieurs composants non-déterministes. Ces composants, décrits aussi en SystemC pour uniformiser les interfaces, utilisent généralement des générateurs aléatoires pour fournir les données nécessaires. Pour tester les aspects temporels, il convient d’intercaler entre les différentes communications des instructions `wait` avec une durée dont la valeur a été générée aléatoirement, ou de préférence intelligemment en fonction du système testé. Nous pouvons imposer que ces durées soient dans un intervalle donné. Dans le cas où l’on génère une durée nulle, il faut bien penser à faire la distinction entre les instructions `yield` et `wait(SC_ZERO_TIME)`. L’instruction `yield`, introduite à la section 4.1.3, a pour effet que le processus l’exécutant rend la main à l’ordonnanceur mais reste éligible. Il se peut qu’une durée ne soit pas théoriquement majorée, mais en pratique il suffit de remplacer l’infini par un nombre suffisamment grand.

8.1.2 Système partiellement temporisé

Historiquement, les premiers modèles TLM développés à STMicroelectronics ne comportaient que des instructions d'attente avec durées fixes. Il pouvait s'agir soit d'une durée d'attente d'un δ -cycle, soit d'une durée fixée plus ou moins arbitrairement. Cependant, certains cas ont montré que pour une plus grande robustesse, les systèmes devaient être testés avec des durées variables.

A première vue, n'avoir aucune information sur le temps dans les systèmes peut sembler être une solution idéale pour garantir la robustesse du système et de son logiciel embarqué.

Techniquement, cela est faisable. Les instructions d'attentes sur du temps, y compris les attentes sur des δ -cycles, doivent être remplacées par des instructions `yield` introduites à la section 4.1.3. Les autres structures ayant recours à des δ -cycles doivent aussi être remplacées par des équivalents non-temporisés. Par exemple, les signaux SystemC `sc_signal` peuvent être remplacés par un simple couple variable - événement, ou par un sous-système plus complexe si l'on souhaite modéliser le fait que la valeur écrite a besoin d'un certain temps pour se propager. Une fois ces transformations faites, toute l'exécution du système se fait en un seul δ -cycle. L'exploration des schémas de temporisation revient alors à explorer les différents ordonnancements, ce qui pourrait se faire avec la méthode décrite aux chapitres précédents.

Malheureusement, il est vite apparu que cette solution ne convenait pas car les systèmes strictement non-temporisés ne sont pas assez contraints et permettent des solutions vraiment irréalistes. Même sans connaître tous les délais de façon précise, certains ordres de grandeur sont connus. Il est par exemple certain que transmettre une interruption par un signal demande moins de temps que transmettre une image complète par le bus. Trouver un ordonnancement qui mène à une erreur mais ne respecte pas les ordres de grandeurs connus sur les délais, n'intéresse pas les développeurs.

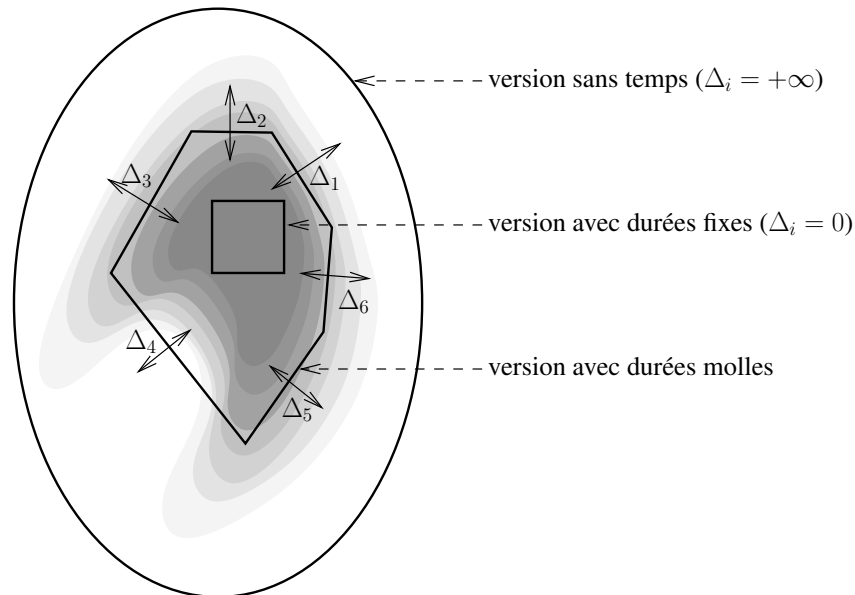


FIG. 8.1 – Ensemble des comportements réalistes de la puce (surfaces grisées), et domaines couverts par les modèles sans temps, temporisés et partiellement temporisés (frontières noires).

Certains délais sont écrits de façon précise et explicite dans la spécification. Par exemple, la durée entre deux rafraîchissements d'un LCD¹ est généralement fixé à $1/24^e$ de seconde. D'autres délais

¹LCD = Liquid Crystal Display (écran)

dépendent de l'implantation, par exemple le temps nécessaire à la décompression d'un bloc d'image JPEG. Dans ces cas là, il est généralement possible de donner un intervalle réaliste en consultant les temps d'exécutions constatés sur des composants matériels analogues.

La solution actuellement en vigueur à STMicroelectronics consiste à utiliser une nouvelle instruction `PV_wait` qui prend en argument un intervalle. L'intervalle est donné sous la forme d'un couple valeur médiane - marge autorisée. Lors de l'exécution, il faut à chaque passage sur une instruction de ce type choisir une valeur dans l'intervalle spécifié et exécuter une instruction `wait` avec la valeur choisie. En augmentant la largeur des intervalles, de nouveaux ordonnancements deviennent possibles, ce qui peut révéler des erreurs de synchronisation. Pour explorer l'ensemble des ordonnancements possibles, modifier les choix de l'ordonnanceur ne suffit plus : il faut aussi agir sur le choix des valeurs pour chaque instruction `PV_wait`. En faisant varier la largeur Δ des intervalles pour chaque délai présent dans le programme, il est possible d'améliorer l'adéquation entre les comportements du modèles, et les comportements (suffisamment) réalistes de la puce finale (cf figure 8.1). Bien sûr, le réalisme d'un jeu de durées n'est pas un concept booléen, mais plutôt une probabilité.

8.1.3 Exemple introductif

L'exemple `chozo` représenté par la figure 8.2 est une variante avec temps imprécis de l'exemple `bozo` de la section 4.2, figure 4.4. Son objectif est de montrer le type d'erreurs qui peut être provoqué par des variations des durées présentes dans le code d'un modèle, ainsi que de montrer les informations qu'il faut parvenir à générer pour valider un modèle.

```

void P() {
    PV_wait(3, d1); // t1
    wait(e);
    PV_wait(40, d2); // t2
    if (x) cout << "Ok\n";
    else cout << "Ko\n";}

void Q() {
    PV_wait(6, d3); // t3
    e.notify();
    x = 0;
    PV_wait(24, d4); // t4
    x = 1;}

```

FIG. 8.2 – Exemple introductif `chozo`

Pour exécuter cet exemple, il faut choisir une durée effective à chaque appel d'une instruction `PV_wait`, en respectant la durée indicative et la marge autorisée : t_1 doit être choisi entre $3-d_1$ et $3+d_1$, t_2 entre $40-d_2$ et $40+d_2$, etc. Dans la suite on utilisera des chaînes de la forme $[t \xrightarrow{+d} T]$ pour indiquer un écoulement du temps d'une durée d menant à la date T .

Si toutes les marges autorisées sont nulles $d_1=d_2=d_3=d_4=0$, alors tous les délais sont fixés et il y a seulement deux exécutions possibles et équivalentes : $P_1; Q_1$ ou $Q_1; P_1$ suivi par $[t \xrightarrow{+3} 3]; P_2; [t \xrightarrow{+3} 6]; Q_2; P_3; [t \xrightarrow{+24} 30]; Q_3; [t \xrightarrow{+16} 46]; P_4$. P_1 et Q_1 s'exécutent à la date $T = 0ns$, P_2 à la date $T = 3ns$, Q_2 et P_3 à $T = 6ns$. Ensuite Q_3 s'exécute à la date $T = 24 + 6 = 30ns$. Enfin, la chaîne "Ok" est affiché par la transition P_4 à la date $T = 6 + 40 = 46ns$.

Pour profiter de la nouvelle instruction `PV_wait`, et ainsi tester la robustesse du programme, il faut donner des valeurs plus grandes aux variables d_i . Si l'on choisit $d_1=d_2=d_3=d_4=2$, il devient alors possible de permuter l'attente et la notification de l'événement e . Il suffit en effet de choisir $t_1 = 5ns$ et $t_3 = 4ns$. Nous retrouvons ainsi le même type de blocage que nous avons obtenu en faisant varier l'ordonnancement de l'exemple `bozo`.

Les valeurs ci-dessus pour les variables d_i ne permettent pas d'obtenir d'autres comportements, à moins de les augmenter encore un peu plus. Supposons que d_2 vaille 10 et que d_4 vaille 6. Il est alors possible de choisir des durées effectives telles que les transitions Q_3 et P_4 s'exécutent à la même

date $T = 6 + 30 = 36ns$ ($30 = 24 + 6 = 40 - 10$). Dans ce cas, le programme devient sensible à l'indéterminisme de l'ordonnancement puisque P_4 peut alors s'exécuter avant Q_3 , causant l'affichage de la chaîne "Ko".

En conclusion, le programme semble fonctionner si l'on se contente d'un seul test avec les durées par défaut, mais des erreurs de conception sont présentes et peuvent se révéler avec d'autres durées réalistes. Étant donné des valeurs pour les variables d_i définissant ce qu'est une durée réaliste, on souhaite avoir un outil générant des jeux de valeurs pour les variables t_i qui permettent de découvrir les erreurs cachées, et garantissant une couverture complète pour un SSTD donné.

8.1.4 Méthodes existantes pour le choix des délais effectifs

L'implantation actuelle de l'instruction `PV_wait` consiste à effectuer un tirage aléatoire dans l'intervalle donné. Toute exécution a donc au moins une chance de se produire mais bien évidemment elle peut être très faible. Une exécution particulière peut être reproduite en fournissant le même germe au générateur pseudo-aléatoire utilisé. Par défaut, un nouveau germe est choisi d'après la valeur de l'horloge du système d'exploitation. En pratique, de nombreuses exécutions redondantes sont générées et d'autres exécutions plus pertinentes restent ignorées.

Quand on lance deux dés, il est bien connu que la valeur 7 ressort plus souvent que la valeur 2 ou la valeur 12. Plus généralement, plus l'on tire des valeurs de façon équiprobable parmi un intervalle donné, plus la moyenne des valeurs tirées se rapproche du milieu de l'intervalle. Bien sûr obtenir une moyenne proche des bornes de l'intervalle reste possible mais la probabilité d'un tel événement devient vite négligeable. Considérez le système représenté par la figure 8.3 : si nous exécutons plusieurs fois ce système avec l'implantation du `PV_wait` actuelle, il est très probable, pour ne pas dire certain, que le processus P terminera toujours avant le processus Q . Cependant, il se peut qu'en obtenant de nouvelles informations, les deux intervalles $[10, 16]$ et $[12, 18]$ se réduisent respectivement en $[14, 16]$ et $[12, 14]$. Dans ce cas aucune des exécutions précédemment générées n'aurait été représentatives de la réalité.

```

void top::P() {
    for (unsigned i=0;i<100;++i) {
        ComputeA();
        PV_wait(13, 3, SC_NS); // [10,16]
    }
    cout <<"P";
}

void top::Q() {
    for (unsigned j=0;j<100;++j) {
        ComputeB();
        PV_wait(15, 3, SC_NS); // [12,18]
    }
    cout <<"Q";
}

```

FIG. 8.3 – Programme utilisant l'instruction `PV_wait`

Cela est un cas assez extrême mais il montre bien qu'un simple tirage aléatoire ne suffit pas si l'on souhaite avoir de bonnes garanties d'avoir exécuté tous les cas réalistes. Un outil automatique ayant recours à des algorithmes plus évolués paraît donc très utile. Étant donné que le principe de base est le même, à savoir chercher à exécuter certain pas de processus avant d'autres, notre outil de génération des ordonnancements semblent une bonne base pour créer un nouvel outil. Ce nouvel outil a pour objectif de générer un sous-ensemble pertinent des divers schémas de temporisations réalistes. Dans la suite de ce chapitre, nous décrivons une version étendue de notre outil, qui génère des délais pour les instructions `PV_wait` ou assimilées, en plus des ordonnancements.

8.2 Principe

Comme pour la génération d'ordonnancements, nous considérons des programmes écrits en SystemC mais qui disposent d'une instruction supplémentaire permettant d'attendre sur un intervalle plutôt que sur une durée précise. L'exécution de cette instruction consiste à choisir une valeur T , dite *durée effective*, dans l'intervalle spécifié puis à appeler la fonction d'attente habituelle : `wait(T)`. Les systèmes utilisant cette instruction seront qualifiés dans la suite de *partiellement temporisés*, par opposition aux systèmes *temporisés* qui ne contiennent que des durées fixées, et aux systèmes *non-temporisés* qui ne contiennent aucune information de temps et s'exécutent intégralement en un δ -cycle.

Plus l'intervalle est large, moins le système est contraint et plus il existe d'exécutions différentes. En effet, augmenter la largeur d'un intervalle peut permettre à un processus de s'exécuter avant ou après un autre processus. L'indéterminisme dû à ces "temps flous" peut donc avoir les mêmes conséquences que l'indéterminisme de l'ordonnanceur. Notre objectif est de garantir que le système fonctionne correctement quelles que soient les durées effectives. Pour cela, nous devons générer un ensemble de *schémas de temporisation* ou *jeux de durées*, c'est-à-dire des durées effectives pour chaque instruction d'attente avec intervalle de temps.

8.2.1 Séparation de l'ordonnement, des durées et des données

Un test comporte plusieurs sortes d'informations : un ordonnancement, des durées effectives et tout le reste que l'on nomme de façon générique "les données", comme représenté par la figure 8.4.

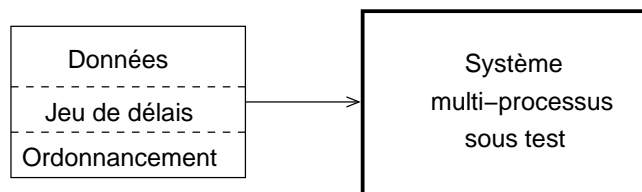


FIG. 8.4 – Un test = des données + des durées + un ordonnancement

Nous supposons de nouveau qu'il est possible de séparer la génération des données, de la génération des ordonnancements et des durées effectives. Pour chaque lot de données, nous générons plusieurs tests complets comportant des jeux de durées différents. Autrement dit, nous testons des couples système sous-test - données, en abrégé SSTD. Notre méthode n'est pas applicable dans le cas où la génération des données n'est pas reproductible, ou qu'elle dépend trop fortement des durées effectives. La méthode présentée dans ce chapitre ne permet pas de dissocier la génération des durées effectives de la génération des ordonnancements.

8.2.2 Les modèles TLM que nous considérons

Certaines fonctionnalités permises par SystemC induisent des difficultés théoriques ou techniques. Nous prenons en compte celles qui sont nécessaires pour traiter les modèles TLM fonctionnels avec temps imprécis. Nous discutons des autres ci-dessous.

8.2.2.1 Limitation sur les δ -cycles

Tout d'abord, nous ne considérons que des programmes qui n'ont qu'un seul δ -cycle entre deux phases d'écoulement du temps. Les programmes avec temps imprécis utilisant le mécanisme des δ -cycles posent à la fois un problème de sémantique et des difficultés techniques.

En effet, la sémantique des délais d'un δ -cycle ne correspond pas à une réalité précise dans le cas des modèles abstraits. Il arrive parfois que des délais d'un δ -cycle soient utilisés pour forcer des ordonnancements ; le résultat est souvent un algorithme peu robuste, comme par exemple l'implantation d'un arbitre décrite à la section 4.4.1. Par ailleurs, les attentes de δ -cycles en présence de temps imprécis simulé par l'instruction `PV_wait` peuvent cacher des ordonnancements qui semblent logiques, et qui seront possibles dans les descriptions plus concrètes du même système. Cela est illustré par l'exemple qui suit.

EXEMPLE 18 — Temps imprécis et δ -cycle

Considérons le petit programme suivant :

```
- processus p : PV_wait(8, 4); cout <<"P";
- processus q : PV_wait(12, 6); wait(SC_ZERO_TIME); cout <<"Q";
                wait(SC_ZERO_TIME); cout <<"Q";
```

Les résultats *PQQ* et *QQP* sont possibles, c'est-à-dire que des timings valides permettent de les obtenir. En revanche, aucun timing ne mène au résultat *QPQ*.

Dans l'exemple, un entrelacement a donc été interdit de façon implicite, et probablement involontaire s'il s'était agit d'un vrai exemple. Or nous devons vérifier que le SSTD fonctionne grâce à des synchronisations volontaires et explicites. Une alternative serait d'autoriser à exécuter un nombre aléatoire d'appels à `wait(SC_ZERO_TIME)` lors d'un appel à `PV_wait`, mais cela est délicat à envisager dans notre contexte à cause du surcoût induit par les multiples changements de contexte.

Dans le cas général, pour dater un événement d'une simulation SystemC, il faut deux informations : une durée, par exemple 0,42 secondes, et le nombre de δ -cycles écoulés depuis la dernière phase d'écoulement du temps. Une date ou une durée devrait donc se représenter avec un couple de $\mathbb{R}^+ \times \mathbb{N}$, et deux dates se trient avec l'ordre lexicographique. Cela est connu dans la littérature sous le nom de temps *super-dense* [LML06]. Notre solution suppose de résoudre des systèmes de contraintes linéaires, dont les variables représentent des durées. Pour cela, les choses seraient grandement compliquées s'il fallait travailler sur l'espace $\mathbb{R}^+ \times \mathbb{N}$, alors qu'avec cette limitation, nous travaillons simplement sur \mathbb{R}^+ .

Sauf cas particulier, il n'est pas de possible de vérifier statiquement qu'un programme n'utilise pas le système des δ -cycles. Cela est au contraire trivial à vérifier sur une trace d'exécution.

8.2.2.2 Limitation des conséquences de la temporisation sur la fonctionnalité

La seconde limitation porte sur l'utilisation des informations temporelles par les processus SystemC. Dans la technique présentée ci-dessous, nous analysons les exécutions pour déduire les impacts des variations de durées sur la fonctionnalité. Cela oblige à enregistrer tous les événements qui sont sensibles à la temporisation.

Parmi toutes les fonctions liées à la temporisation, les plus importantes sont l'instruction `wait` classique, et sa nouvelle version imprécise `PV_wait`. Jusqu'à présent, ce sont les seules que nous avons rencontrées dans les études de cas, et en conséquences ce sont les seules que nous traitons pour le moment.

Une autre fonction dont le comportement dépend du temps est bien évidemment la fonction `sc_time_stamp` qui retourne la “date” courante (variable t de la figure 4.2). Notre solution n’est pas compatible avec une utilisation générale de cette fonction. Elle pourrait juste être complétée pour les cas où cette fonction est utilisée dans une contrainte linéaire, par exemple `if (sc_time_stamp() < K) { . . . }`. Il n’est par contre pas possible de traiter une affectation du type `x = black_box(sc_time_stamp())`, car cela peut induire un ensemble infini de comportements fonctionnels différents.

L’usage de cette fonction n’a pas besoin d’être pris en compte pour la validation tant qu’elle est utilisée uniquement pour les affichages de débogue, ou d’une façon qui n’a pas d’impact sur la fonctionnalité. Il est possible de vérifier que l’on se trouve dans ce cas-là, en vérifiant simplement que toutes les occurrences de cette fonction sont dans du code d’affichage ou assimilé. Ce cas est cohérent avec notre contexte, où les annotations de durées sont ajoutées à des modèles fonctionnels, dans le seul but d’éviter les comportements physiquement irréalistes.

8.2.3 Aperçu général

Le principe général reste le même : nous exécutons le SSTD avec des durées et un ordonnancement quelconques, puis nous examinons en détail les communications qui ont eu lieu pour générer de nouvelles valeurs susceptibles de mener à un état final différent. Nous recommençons ensuite itérativement sur chaque nouvelle exécution pour générer de proche en proche tout un ensemble de jeux de durées. Chaque jeu de durées généré est accompagné d’un ordonnancement, ou plus exactement de contraintes permettant la génération d’un ordonnancement.

Quand toutes les durées sont fixes, il n’est possible que de permuter des transitions (c’est-à-dire un pas d’exécution d’un processus, cf section 5.3) d’un même δ -cycle. Modifier les durées effectives permet d’envisager d’autres permutations. Pour chaque couple de transitions, il faut d’une part regarder s’il est **utile** de les permuter, et d’autre part si cela est **possible**. Concernant l’utilité rien ne change par rapport à la génération des ordonnancements : une permutation est utile si les transitions concernées contiennent un couple d’actions de communication qui donneraient des résultats différents si elles étaient exécutées dans l’autre ordre. Seule la question de la possibilité d’une permutation est à réexaminer.

Un système partiellement temporisé peut être vu comme un système temporisé dans lequel certaines contraintes ont été assouplies. A la section 4.1, nous avons dit qu’une exécution était découpée en plusieurs cycles séparés par des phases de mises à jour et d’écoulement du temps. Vu comme cela, modifier une durée revient à déplacer une transition d’un cycle à un autre. Une approche aurait pu être d’étudier la répartition des transitions sur les différents cycles, puis d’exécuter l’outil de répartition des ordonnancements sur chaque cycle. Cependant, nous avons choisi l’approche inverse : nous considérons un système partiellement temporisé comme un système non-temporisé auquel des contraintes ont été ajoutées. Cette approche nous a semblé plus simple et plus efficace, mais ce dernier point ne reste encore qu’une supposition.

L’analyse d’une trace d’exécution dans le but de trouver d’autres comportements se fait en deux étapes. La première étape utilise la technique de réduction d’ordre partiel précédemment décrite ; la seconde utilise des techniques de programmation linéaire. Considérer un système partiellement temporisé comme un système non-temporisé signifie que, dans un premier temps, nous ignorons tous les changements de cycles. Nous fournissons à l’outil de calcul des dépendances la trace d’exécution, qui liste les actions exécutées par chacune des transitions et certaines informations sur les synchronisations survenues (activation d’un processus), mais qui ne comporte plus d’informations liées au temps ou aux changements de cycles. Cette étape permet d’obtenir un graphe des dépendances dy-

namiques comme celui de la figure 8.5. La signification des symboles est globalement la même qu'à la sous-section 5.3.3. Il y a juste deux nouveautés : d'une part les attentes sur des durées variables sont représentées par des lignes courbes ou "ressort", d'autre part les changements de cycle sont représentés par des barres verticales pointillées puisqu'ils ne correspondent plus à des barrières infranchissables par les transitions.

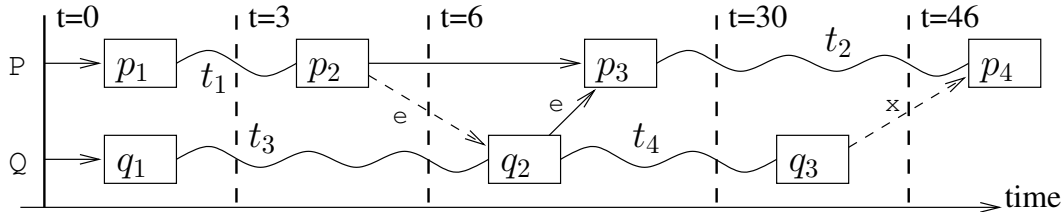


FIG. 8.5 – Graphe des dépendances dynamiques pour l'exemple `chozo`, avec le timing $t_1 \mapsto 3$, $t_2 \mapsto 40$, $t_3 \mapsto 6$, $t_4 \mapsto 24$, et l'ordonnancement $p_1q_1p_2q_2p_3q_3p_4$.

L'outil de calcul des dépendances fournit des paires de transitions qui sont dépendantes et qui semblent permutable. Plus précisément, leur permutation n'est pas empêchée par des synchronisations explicites, mais peut l'être par les contraintes temporelles que l'on n'a pas encore prises en compte. Le rôle de la deuxième étape est justement de déterminer s'il existe un jeu de durées valides permettant cette permutation, et si oui de fournir une solution particulière afin de pouvoir ré-exécuter le SSTD.

Considérons pour exemple la trace d'exécution et les dépendances représentées par le graphe de la figure 8.5. Le code des processus est donné par la figure 8.2 et on suppose que $d_1=d_3=2$, $d_2=10$ et $d_4=6$. Il y a deux flèches pointillées désignant des transitions dépendantes et à priori permutable. Pour chacune, nous allons construire un système de contraintes tel que les éventuelles solutions de ce système soient des jeux de durées valides permettant la permutation visée.

La première flèche correspond à la contrainte d'ordonnements $q_2 < p_2$. Pour la valider, il faut que t_3 soit plus petit que t_1 . Si $t_1 = t_3$, alors l'ordonneur modifié précédemment décrit permet de garantir cette contrainte d'ordonnement. Grâce à cela, nous n'aurons jamais besoin de contraintes strictes. Par ailleurs, nous devons respecter les bornes sur les délais telles que définies par les paramètres fournis lors de l'appel à l'instruction `PV_wait`. Dans notre cas : $t_1 \in [1, 5]$ et $t_3 \in [4, 8]$. Nous obtenons ainsi le système $t_3 \leq t_1 \wedge t_1 \in [1, 5] \wedge t_3 \in [4, 8]$. Ce système a une infinité de solutions : il suffit de choisir $t_1 \in [4, 5]$ puis $t_3 \in [4, t_1]$. Chacune de ces solutions conduit au blocage causé par la mauvaise utilisation de l'événement `e`, à condition de ne pas oublier la contrainte d'ordonnement pour le cas où $t_1 = t_3$.

Pour la deuxième paire de transitions dépendantes et candidates à la permutation, il faut trouver un jeu de durées compatible avec les contraintes d'ordonnement $\{p_2 < q_2, p_4 < q_3\}$. Pour la première, il suffit d'inverser la première contrainte du système précédent : $t_3 \geq t_1 \wedge t_1 \in [1, 5] \wedge t_3 \in [4, 8]$. Dans le cas présent, la spécification de SystemC impose que p_3 et q_2 s'exécutent à la même date car la notification est immédiate et le temps ne s'écoule pas tant qu'il reste au moins un processus éligible. Il suffit donc d'avoir t_2 plus petit que t_4 . En ajoutant les contraintes issues des bornes des durées, on obtient le système : $t_3 \geq t_1 \wedge t_1 \in [1, 5] \wedge t_3 \in [4, 8] \wedge t_2 \leq t_4 \wedge t_2 \in [30, 50] \wedge t_4 \in [18, 30]$. Il y a de nouveau une infinité de solutions : il suffit de choisir $t_1 \in [1, 5]$ puis $t_3 \in [\max(t_1, 4), 8]$, et enfin $t_2 = t_4 = 30$. Toutes ces solutions, accompagnées des deux contraintes d'ordonnement, conduisent à l'affichage de la chaîne "Ko".

Nous avons ainsi mis en évidence les trois comportements possibles de l'exemple étudié.

8.2.4 Algorithme formel

Nous décrivons maintenant comment faire cela dans le cas général et automatiquement.

Afin de formaliser la suite de la présentation, nous associons un identifiant $\omega \in \Omega$ à chaque instruction `PV_wait` (D, d). Nous notons $B(\omega)$ (B comme *Bornes*) l'intervalle $[D - d, D + d]$, et $\#_u(\omega)$ le nombre d'exécutions d'une instruction `PV_wait` particulière pour un ordonnancement u . Un *timing* ou *jeu de durée* est une fonction des paires $(\omega, n) \in \Omega \times [1.. \#_u(\omega)]$ vers des durées d . $T(\omega, n) = d$ signifie que l'on attendu une durée effective d lors du n -ème appel de l'instruction `PV_wait` identifié par ω . Par définition, un jeu de durées T est valide si et seulement si $\forall (\omega, n) \in \Omega \times [1.. \#_u(\omega)], T(\omega, n) \in B(\omega)$.

La date effective d'une transition est la valeur de la variable t de la figure 4.2 lorsqu'elle est exécutée. Contrairement aux chapitres précédents, la date d'une transition peut varier d'une exécution à l'autre même si les données et l'ordonnancement reste constant. Par contre, le comportement fonctionnel restent le même, grâce aux limitations actuelles concernant l'accès à la date globale par le processus.

8.2.4.1 Génération des contraintes temporelles

A l'issu de l'analyse des dépendances d'une trace d'exécution, on obtient des paires de transitions dépendantes qu'il faut essayer de permuter. A chaque paire correspond un ensemble de contraintes d'ordonnements : une pour la paire elle même, une pour chaque paire précédente, et d'autres provenant des exécutions et analyses précédentes. Nous montrons ici comment ces contraintes d'ordonnement se traduisent en inégalités numériques représentant les contraintes de temporisation. Les variables de ces inégalités correspondent aux durées effectives qu'il faut choisir à chaque exécution d'une instruction `PV_wait`. La conjonction de ces contraintes d'ordonnement donne un *système linéaire* (ou *programme linéaire*) que l'on sait résoudre de façon exacte. Ces solutions sont des jeux de durées permettant d'appliquer les contraintes d'ordonnement correspondantes.

La première étape consiste à définir une *date symbolique* $sdate(p_i)$ pour chaque transition p_i . Cette date symbolique est une expression linéaire qui représente la date de la transition en fonction des variables de durées $T(\omega, n)$. Elle est définie récursivement à partir de la date symbolique des transitions précédentes.

Définition 19 — Date symbolique

Étant donné un ordonnancement u , la date symbolique d'une transition p_i de u vaut :

1. $sdate(p_i) = 0$ si $i = 1$ et le processus p est initialement éligible ;
2. $sdate(p_i) = sdate(q_j)$ si la transition p_i a été activée par $q_{j,u}$ (notification immédiate) ;
3. $sdate(p_i) = sdate(p_{i-1}) + T(\omega, n)$ si la transition p_{i-1} s'est terminée par le n -ème appel à l'instruction `PV_wait` identifiée par ω ;
4. $sdate(p_i) = sdate(p_{i-1}) + D$ si la transition p_{i-1} s'est terminée par un appel à une instruction `wait` (D) (D est une constante dont on connaît la valeur).

Dans le cas d'un appel à `PV_wait` (D, d) avec $d = 0$, ce qui sémantiquement est équivalent à un simple `wait` (D), on peut noter que la règle 3 est bien cohérente avec la règle 4 puisque par ailleurs $T(\omega, n) \in [D, D]$.

EXEMPLE 19 — Dates symboliques pour un ordonnancement de `chozo`

On considère l'exemple `chozo` (figure 8.2) avec l'ordonnancement $u = p_1q_1p_2q_2p_3q_3p_4$.

- $sdate(p_1) = sdate(q_1) = 0$ d'après la règle 1 ci-dessus ;

- $sdate(q_2) = t_3$, $sdate(p_2) = t_1$ et $sdate(q_3) = t_3 + t_4$ d'après la règle 3 ;
- $sdate(p_3) = sdate(q_2) = t_3$ d'après la règle 2 ;
- $sdate(p_4) = sdate(p_3) + t_2 = t_3 + t_2$ d'après la règle 3.

Si on applique le jeu de durées $t_1 \mapsto 3, t_2 \mapsto 40, t_3 \mapsto 6, t_4 \mapsto 24$, on constate que l'on obtient bien les mêmes dates effectives que dans le graphe des dépendances dynamiques (figure 8.5).

—

Nous pouvons maintenant associer des contraintes temporelles aux contraintes d'ordonnancement.

Définition 20 — Contrainte temporelle

Étant donné un ordonnancement u , la contrainte temporelle d'une contrainte d'ordonnancement " $p_i < q_j$ ", noté $TC("p_i < q_j")$ est une inégalité numérique obtenue :

1. en commençant par l'expression $sdate(p_{i,u})$;
2. puis en plaçant un signe " \leq " ;
3. et en terminant par l'expression $sdate(q_{j,u})$.

8.2.4.2 Algorithme principal

La figure 8.6 présente le nouvel algorithme principal. La structure générale est semblable à celle de l'algorithme pour la génération des ordonnancements seuls (figure 5.6).

```

 $G_T$ (contraintes d'ordonnements  $C$ , durées  $T$ ) : //appel initial :  $G_T(\emptyset, \emptyset)$ 
  exécuter le SSTD en respectant les contraintes  $C$  et les durées  $T$  ; (1)
   $u$  = ordonnancement de l'exécution ci-dessus ;
  système linéaire  $S = []$  ;
  pour tous  $(\omega, n) \in \Omega \times [1.. \#(\omega)]$  faire :
     $S = S \bullet (T(\omega, n) \in B(\omega))$  ;
  pour toutes les contraintes " $p_i < q_j$ " de  $C$  faire :
     $S = S \bullet (sdate(p_i) \leq sdate(q_j))$  ;
  pour toutes les paires de transitions  $p_i$  et  $q_j$  de  $u$  tel que  $p_i <_u q_j$ 
    et  $(p_i, q_j) \in \mathcal{D}' \cap \mathcal{P}' | C$  faire : (2)
    si is_feasible( $S \bullet TC("q_j < p_i")$ ) alors (3)
       $T' = \text{solution\_of}(S \bullet TC("q_j < p_i"))$  ;
       $G_T(C \cup "q_j < p_i", T')$  ; (4)
       $C = C \cup "p_i < q_j"$  ;
       $S = S \bullet TC("p_i < q_j")$  ;

```

FIG. 8.6 – Algorithme principal pour la génération conjointe des ordonnancements et des jeux de durées.

En plus d'un ensemble de contraintes d'ordonnancement, l'algorithme G_T prend un jeu de durées en argument. Celui-ci est utilisé à la ligne (1) pour permettre une exécution du SSTD compatible avec les contraintes d'ordonnancement. Ce jeu de durées n'est en général que partiel, c'est-à-dire que la valeur de certains $T(\omega, n)$ peut être inconnu. Dans ce cas, une durée est choisie aléatoirement dans l'intervalle spécifié par les paramètres de l'instruction `PV_wait`.

Le système linéaire S est initialisé, d'une part avec les contraintes sur les bornes des variables $T(\omega, n)$, et d'autre part à partir des contraintes d'ordonnements héritées des exécutions précédentes (ligne (4)). Le système est ensuite complété parallèlement à l'ensemble de contraintes

d'ordonnement. Les relations $\mathcal{P}'|C$ et \mathcal{D}' (ligne (2)) sont calculées sans tenir compte du temps et des changements de cycle. Le jeu de durées effectif T_u est, à toutes les étapes, une solution du système S . En général, il n'est pas une solution du système construit à la ligne (3), sauf s'il se trouve sur la nouvelle contrainte (cf figure 8.7).

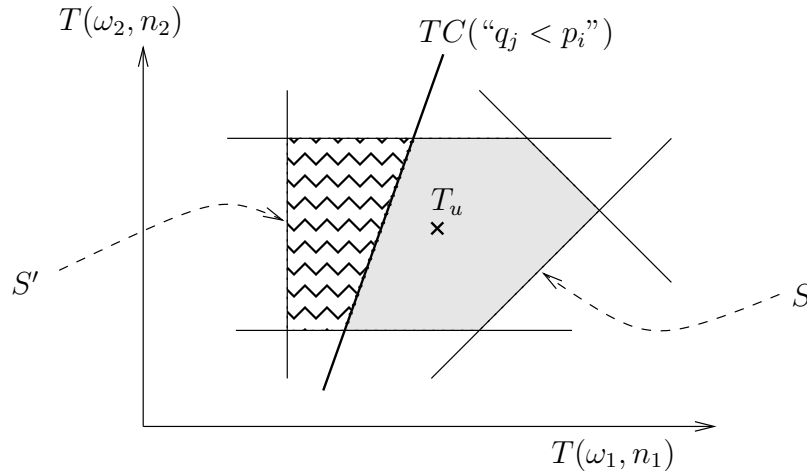


FIG. 8.7 – Coupure des nouveaux systèmes lors de l’ajout d’une contrainte d’ordonnement. S' correspond au système construit à la ligne (3).

Les systèmes d’inégalités générés étant linéaires, les fonctions “is_feasible” et “solution_of” peuvent être implantées en utilisant les techniques de programmation linéaire, par exemple l’algorithme du Simplex. Avec la sémantique actuelle de l’instruction `PV_wait`, les systèmes générés sont des octaèdres (tous les coefficients sont dans $\{-1, 0, 1\}$, cf [CC04]), mais pas des octogones (une contrainte peut impliquer plus de deux variables, cf [Min01]).

Afin d’illustrer l’algorithme ci-dessus, nous décrivons ci-dessous le premier appel à G_T pour l’exemple `chozo`. On suppose que l’ordonnement et les durées effectives de la première exécution sont celles de la figure 8.5. La première étape consiste à calculer les dépendances pour obtenir des ensembles de contraintes d’ordonnement. Ici, nous obtenons $\{q_2 < p_2\}$ et $\{p_2 < q_2; p_4 < q_3\}$.

Le premier ensemble de contraintes $\{q_2 < p_2\}$ donne un système linéaire S' qui contient uniquement la contrainte $sdate(q_2) \leq sdate(p_2)$ qui se réécrit en $t_3 - t_1 \leq 0$, et les bornes $t_1 \in [1, 5]$ and $t_3 \in [4, 8]$. L’algorithme G_T demande une solution à une librairie de programmation linéaire et obtient par exemple la solution $t_1 = t_3 = 4$. Il appelle alors $G_T(\{q_2 < p_2\}, \{t_1 = 4, t_3 = 4\})$ (ligne (4)). Grâce à cette contrainte d’ordonnement et ce jeu de durées, l’interblocage décrit à la section 8.1.3 est découvert.

Le deuxième ensemble de contraintes $\{p_2 < q_2; p_4 < q_3\}$, donne, en plus des bornes des variables, les deux contraintes $t_3 - t_1 \geq 0$ et $t_2 - t_4 \leq 0$. Une solution est par exemple $t_1 = t_3 = 4$ et $t_2 = t_4 = 30$. Enfin, G_T est appelé avec cet ensemble de contrainte et ce jeu de durées, ce qui révèle la seconde erreur décrite à la section 8.1.3.

8.2.4.3 L’ensemble de jeux de durées généré

Comme pour G_S , l’algorithme G_T génère **au moins un** représentant de chaque classe d’équivalence s’il est exécuté jusqu’à son terme. On peut déjà noter que c’est bien le cas sur l’exemple `chozo` qui compte 3 classes d’équivalence.

On définit un troisième algorithme G'_S . Par rapport à G_S , la seule différence est que l'on ne tient plus compte du temps et des changements de cycle. Notamment, les relations $\mathcal{P}'|C$ et \mathcal{D}' (ligne (2)) sont calculées sans tenir compte du temps et des changements de cycle, comme pour G_T .

G'_S (contraintes d'ordonnements C) : //appel initial : $G'_S(\emptyset)$
 exécuter le SSTD en respectant les contraintes C ;
 $u =$ ordonnancement de l'exécution ci-dessus ;
 pour toutes les paires de transitions p_i et q_j de u tel que $p_i <_u q_j$
 et $(p_i, q_j) \in \mathcal{D}' \cap \mathcal{P}'|C$ faire : (2)
 $G'_S(C \cup "q_j < p_i")$;
 $C = C \cup "p_i < q_j"$;

FIG. 8.8 – Algorithme G'_S (rappel de l'algorithme G_S , seules les relations \mathcal{D} et \mathcal{P} changent).

Étant donné un SSTD, Ce nouvel algorithme renvoie le même résultat :

- que l'algorithme G_S si on remplace dans le SSTD toutes les instructions `wait` et `PV_wait` par une instruction `yield` ;
- ou que l'algorithme G_T si on remplace dans le SSTD toutes les instructions `yield`, `wait` et `PV_wait` par une instruction `PV_wait` avec bornes infinies.

D'après la preuve pour G_S , l'algorithme G'_S génère théoriquement au moins un représentant de chaque classe d'équivalence. En pratique, cet ensemble est généralement trop gros pour permettre l'exécution de G'_S jusqu'à son terme.

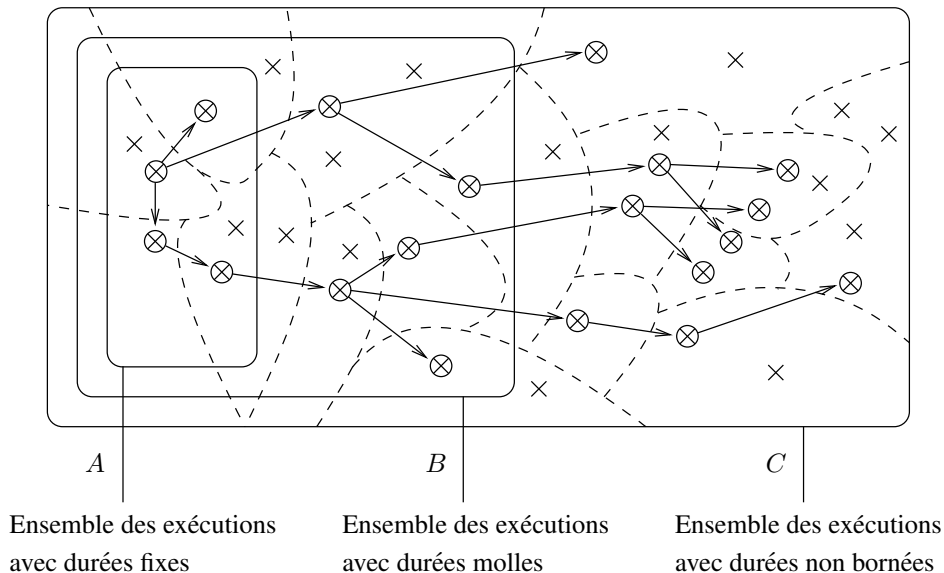


FIG. 8.9 – Ensemble de toutes les exécutions d'un SSTD. Les lignes pointillées délimitent les classes d'équivalence. Les croix représentent des exécutions valides ; celles qui sont entourées désignent les exécutions générées. Les flèches relient les exécutions "mères" aux exécutions "filles". L'algorithme G_S retourne l'ensemble des croix entourées de l'ensemble A , G_T celles de l'ensemble B et G'_S celles de C .

Par rapport à G'_S , G_T contient les instructions pour coder les contraintes temporelles sous forme de

système linéaire, et pour tester leur faisabilité. Nous savons par construction qu'il existe une exécution (u, T) qui satisfait un ensemble de contraintes d'ordonnancement C , si et seulement si le système S construit à partir de C a au moins une solution. Ainsi, G_T génère tous les éléments qui satisfont les contraintes temporelles, parmi ceux qu'aurait générés G'_S . La figure 8.9 représente les ensembles d'exécutions générés par G_S , G'_S et G_T .

8.3 Implantation

L'architecture globale (figure 8.10) reste inchangée par rapport au prototype précédent. L'instruction `PV_wait` est instrumentée de façon à compléter la trace d'exécution avec des éléments de la forme `<rvt_wait min="18" max="30"/>`. Les seules modifications conséquentes se situent au niveau de l'analyseur, où nous devons construire puis résoudre des systèmes de contraintes linéaires, puis enregistrer les résultats. Le nouvel outil, correspondant à l'algorithme G_T , a été baptisé `rvt`.

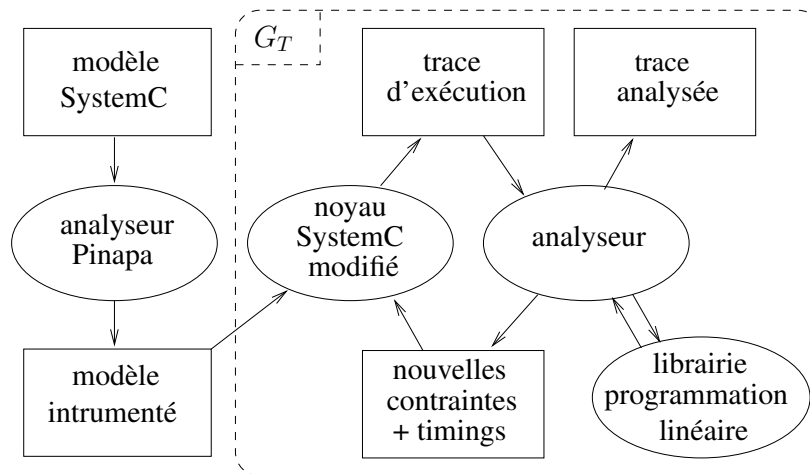


FIG. 8.10 – Architecture globale pour la génération de schémas de temporisation.

8.3.1 Construction des contraintes linéaires

Nous devons construire une contrainte linéaire pour chaque contrainte d'ordonnancement hérité, ainsi que pour chaque contrainte d'ordonnancement généré. Les bornes sur les variables sont stockées à part. Les contraintes linéaires sont construites sous forme de liste de coefficients et d'une constante. Par exemple $x_1 + x_3 - x_4 \leq 42$ se code sous la forme $([1, 0, 1, -1], 42)$. Les coefficients des contraintes découlent directement de la définition des dates symboliques.

En dehors des transitions initialement éligibles, la date symbolique d'une transition a est toujours la somme de la date symbolique d'une transition précédente b , que l'on appellera ici *référence temporelle de a* , et d'une variable (temporelle) ou d'une constante éventuellement nulle. Une structure de donnée additionnelle, `Tsteps` de profil $transition \rightarrow transition \times variable \times constante$, associe à chaque transition sa référence temporelle (identifiée ici par un numéro de pas d'exécution). Les deux champs suivants sont l'index de la variable et la constante, que nous devons ajouter le cas échéant, mais il y a un décalage : la variable et la constante d'une transition sont en réalité associées dans cette structure à sa référence temporelle. Ce décalage permet de stocker l'information

dès que nous la rencontrons ; la durée qu'une transition a attendue (ou l'intervalle), est en effet connue à la fin de la transition de référence, c'est-à-dire celle qui a exécuté l'instruction `wait` (ou `PV_wait`). Cette nouvelle structure de données est mise à jour au début de chaque transition rencontrée dans la trace analysée. Une nouvelle variable est créée chaque fois que l'on rencontre une balise `<rvt_wait ...>`. Le code ci-dessous permet de calculer la date symbolique d'une transition identifiée par son numéro de pas.

C : tableau de coefficients ; cst : constante

Initialement, les coefficients et la constante sont nulles.

s est le numéro de la transition sujette.

```
while (s>0) { // Le calcul se fait de la droite vers la gauche.
    p = Tsteps[s].prec; // p désigne la référence temporelle
    if (Tsteps[p].var>=0) // L'index -1 indique qu'aucune variable n'est à ajouter.
        C[Tsteps[p].var] = +1; // p est utilisé à la place de s à cause
    cst += Tsteps[p].duration; // du décalage présent dans la structure
    s = p; // Les itérations suivantes calculeront la date symbolique de la référence.
}
```

En pratique, nous calculons directement la différence des deux dates symboliques qui forme la contrainte. Cela est plus efficace car dès que l'on rencontre une référence temporelle commune aux transitions, nous savons que tous les coefficients des variables d'index inférieurs seront nulles.

8.3.2 Résolution des systèmes linéaires

Les implantations des algorithmes de résolution de systèmes linéaires sont nombreuses. La seule difficulté consiste ici à choisir la librairie la plus adaptée à notre cas. Nous avons pensé à CLP (*COIN-OR Linear Program Solver*) [clp04], GLPK (*GNU Linear Programming Kit*) [glp00] et LP_SOLVE [B⁺96]. Nous avons finalement opté pour LP_SOLVE car elle nous a semblé simple d'utilisation, et sa licence LGPL peut s'avérer très intéressante si STMicroelectronics décide de diffuser l'outil ici décrit.

8.3.3 Représentation des schémas de temporisation

Nous obtenons les résultats sous forme d'une liste de durées, qui correspondent chacune à une variable temporelle. Étant donné deux exécutions équivalentes, les appels à l'instruction `PV_wait` pour un processus particulier sont toujours les mêmes, et dans le même ordre. Par exemple, si le processus *p* a exécuté d'abord un `PV_wait` à la ligne 35 puis un second à la ligne 12, alors dans toutes les exécutions équivalentes, ce même processus *p* exécutera toujours le `PV_wait` de la ligne 35 en premier et celui de la ligne 12 en second. Grâce à cela et aux contraintes d'ordonnements associées, il n'est pas nécessaire pour l'implantation d'identifier chaque appel d'instruction `PV_wait` par un label ω_n , tel que c'est fait dans la théorie. Pour ré-exécuter un jeu de durées en présence des contraintes d'ordonnements ayant permis sa génération, il nous est suffisant d'associer chaque durée au processus qui a appelé le `PV_wait` correspondant. Au final, les solutions sont enregistrées dans des fichiers où chaque ligne comporte un couple : identifiant de processus - durée. Lors de la ré-exécution du SSTD, la fonction `PV_wait` va consommer dans l'ordre les valeurs associées au processus courant, tant que cela lui est possible.

8.4 Évaluation

Pour l'évaluation de notre nouveau prototype, nous reprenons un modèle déjà étudié avec l'outil précédent : le décodeur LCMPEG. Il s'agit d'un modèle fonctionnel de taille moyenne fourni par STMicroelectronics. Sa description est disponible à la section 7.3.

En remplaçant les instructions d'attente sur du temps `wait(d)` par la nouvelle instruction `PV_wait(d, R*d)`, nous obtenons un modèle partiellement temporisé, dont nous pouvons relâcher les contraintes temporelles en augmentant la constante globale R . Plus les contraintes temporelles sont lâches, plus les ordonnancements valides sont nombreux. L'objectif est de valider le modèle avec R le plus grand possible.

Nous avons réussi à valider le modèle avec $R = 0.2$. Notre outil `rvt` a généré **3584 ordonnancements et schémas de temporisation**. Cela lui a demandé **35 minutes et 11 secondes**, qui se répartissent en 23 min 18 sec pour les 3584 simulations, et 11 min 53 sec pour les calculs annexes, soit environ un tiers du temps total. L'utilisation des utilisations des événements persistants n'a pas amélioré ce résultat (cf section 7.3.4).

Il est aussi possible d'obtenir une version strictement non temporisée de ce modèle. Il suffit pour cela de remplacer les instructions `wait(d)` par notre instruction `yield()` introduite à la section 4.1.3. Sémantiquement, cela est équivalent à n'utiliser que des instructions `PV_wait` avec des intervalles de taille infinie. Il est théoriquement possible de valider ce nouveau modèle avec l'outil `rvs`. En pratique, l'espace d'état à parcourir est beaucoup trop grand. L'analyse de la première trace d'exécution révèle 32 dépendances permutable. Il faut donc s'attendre à devoir lancer 2^{32} exécutions avec autant d'ordonnements différents, ce qui demanderait plusieurs années de simulations.

En conclusion, la validation de modèles partiellement temporisés est beaucoup plus coûteuse que la validation de modèles avec temps fixes. Contrairement à ce que nous avons craint au début, le surcoût dû à la génération et résolution des contraintes linéaires est très acceptable, bien que cette partie ait été codée de façon très naïve. Le vrai problème est la forte croissance du nombre d'ordonnements à considérer. En conséquence, les meilleures optimisations sont à chercher dans l'algorithme de réduction d'ordre partiel, c'est-à-dire dans la partie commune avec `rvs`. Pour le moment, nous sommes capables de valider des modèles de taille moyenne avec des ratios qui commencent à être intéressants.

8.5 Autres approches pour la validation de modèles avec temps imprécis

L'idée qui consiste à interpréter les informations de façon non stricte est assez naturelle. Cela était déjà présent dans certaines approches de modélisation basées sur le temps imprécis (voir, par exemple, [LAK98]). La seule nouveauté est son intégration au flot de développement des modèles TLM. La part innovante de ces travaux se trouve donc essentiellement dans la façon de les valider.

8.5.1 Test, plus réduction d'ordre partiel ou programmation linéaire

L'approche décrite dans [YKM02] a certaines similarités avec la nôtre. Ils utilisent leur outil de vérification VINAS-P sur le programme sous test qui comporte des délais bornés. Cela leur retourne des cas de tests qui exhibent des *fautes*. Cette première étape utilise des réductions d'ordre partiel statiques [YR99]. Ensuite, pour chaque trace menant à une faute, ils génèrent un ensemble de contraintes linéaires, qu'ils résolvent ensuite avec un solveur de programmes linéaires en nombres entiers (ILP).

La solution de ces systèmes leur fournit de nouvelles bornes pour les délais du programme sous test, qui permettent d'éviter les fautes précédemment détectées. Comme pour nos expérimentations, ils concluent que le temps nécessaire pour la résolution des contraintes linéaires est très acceptable par rapport au temps total. La technique utilisée dans notre outil diffère en trois points importants :

- les réductions d'ordre partiel que nous utilisons sont dynamiques ;
- la programmation linéaire est directement combinée avec la réduction d'ordre partiel, alors que chez eux elle est utilisée après coup ;
- les programmes considérés sont des descriptions de circuits à très bas niveau (niveau porte).

La programmation linéaire avait déjà été utilisée dans le même contexte auparavant, mais sans réduction d'ordre partiel [HG96].

Nous avons expliqué dans l'introduction que notre outil pouvait aussi servir à couvrir les délais possibles pour les générateurs d'entrées, c'est-à-dire les délais de la spécification. [NS01] décrit une méthode pour générer des tests temporisés à partir d'une spécification formelle (sous forme d'automates temporisés). Les contraintes temporelles sont mises sous la forme de *zones* (DBM², ce qui est plus spécifique que les octaèdres et les octogones). Ils choisissent ensuite des points dans ces octogones pour *couvrir la spécification*, tout en évitant de générer des tests *équivalents*. Comme toute approche de génération basée sur la spécification, elle est en principe indépendante du langage de l'implantation. [CL97] propose aussi de générer des tests en fonction de contraintes temporelles décrites par une spécification graphique.

8.5.2 Extraction d'un modèle puis vérification formelle

Nous ne connaissons pas d'outil de vérification formelle pour programmes SystemC avec délais bornés. Les outils de vérification travaillent généralement sur des langages formels de bas niveau (i.e. proches des modèles de calculs). Une solution est donc de traduire les modèles TLM dans l'un de ces langages formels. Les automates temporisés constitueraient une solution intéressante, car les intervalles définies par les instructions `PV_wait` peuvent y être directement reportés, soit sur les gardes des transitions, soit dans les invariants des états. Il est souhaitable que la traduction soit automatique car une traduction *manuelle* serait source d'erreurs, ou au mieux incertaine. L'outil LusSy (cf section 3.1) est capable de transformer automatiquement des programmes SystemC en automates synchrones, mais ne prend pas en compte les informations temporelles. Une idée serait de modifier LusSy, pour qu'il génère des automates temporisés, vérifiables avec des outils comme Kronos [BDM⁺98] ou Uppaal [Upp06].

Pour réduire le problème de l'explosion combinatoire, adapter les techniques de réductions d'ordre partiel aux systèmes temporisés est nécessaire. Ce problème a été attaqué dans [Pag96], ainsi que par [BJLY98] qui propose une méthode basée sur une sémantique de *temps locaux*. Il y a eu des évolutions récentes : [LNZ05, Zen04] présentent une nouvelle *sémantique d'ordre partiel* pour les automates temporisés. Enfin, [BBM06] a montré comment intégrer de la réduction d'ordre partiel dans le flot de vérification du langage IF (cf [BFG⁺00]), grâce à la constatation que l'union des zones associées à des chemins équivalents (selon l'analyse des dépendances) est aussi une zone.

Les programmes linéaires que nous générons définissent des octaèdres. Ceux-ci ont aussi été étudiés en détail dans le cadre de la vérification formelle par interprétation abstraite de circuits temporisés, représentés par des réseaux de pétri avec délais sur les transitions ([CC05, Vil05]). Ces travaux n'ont pas recours à des réductions d'ordre partiel.

Malgré toutes ces optimisations, il est probable que les outils de vérifications formelles passent

²DBM = Difference Bound Matrice

moins bien à l'échelle que notre approche qui est basée sur des jeux de tests, que nous complétons automatiquement avec des schémas de temporisation. De plus, notre approche évite le problème du retour aux sources. C'est-à-dire : associer chaque erreur trouvée dans le modèle abstrait, à une erreur dans le code source du programme en cours de validation.

Chapitre 9

Vers un simulateur SystemC parallèle pour modèles TLM

Sommaire

9.1	Objectif et contraintes	139
9.1.1	Le modèle d'exécution actuel de l'OSCI	140
9.1.2	Parallélisation : aperçu général	141
9.1.3	Parallélisation : respect de la spécification	141
9.2	Cadre formel : dépendances pour la parallélisation	143
9.2.1	Définitions : transitions, actions et exécutions parallèles	143
9.2.2	Relation d'indépendance pour la parallélisation	143
9.3	Outils existants, approche structurelle	145
9.3.1	Les modèles SystemC avec communications globalement synchrones	145
9.3.2	Outils existants	146
9.3.3	Relâchement de la non-préemptivité dans le cas des transactions	147
9.4	Vers un outil adapté au TLM, approche non-structurelle	148
9.4.1	Les granularités envisageables	148
9.4.2	Sketch d'ordonnanceur multiprocesseur	150
9.4.3	Analyses statiques pour le pré-calcul des dépendances	151
9.4.4	Identification dynamique des transitions	152
9.5	Perspectives	152

Les travaux présentés dans ce chapitre ont été réalisés en collaboration avec Youssef Bouzouzou et Pascal Raymond.

9.1 Objectif et contraintes

L'utilisation de modèles fonctionnels TLM pour le développement du logiciel embarqué repose sur 2 arguments majeurs : la rapidité pour les écrire et la vitesse de simulation. Le premier point implique que les développeurs du logiciel embarqué pourront commencer les simulations avant que le code RTL, plus complexe, soit écrit. Le deuxième point améliore la productivité de ces développeurs et de l'équipe de validation du futur système sur puce. Notre objectif est ici d'améliorer le deuxième point sans que cela soit au détriment du premier.

Pour cela, l'idée est de **profiter des machines multiprocesseurs**, et des processeurs multi-cœurs, qui deviennent de plus en plus fréquents. Comme SystemC a été conçu pour décrire des systèmes intrinsèquement parallèles, il est probable que leurs simulations soient aussi réalisables en parallèles.

La contrainte majeure est de **respecter** scrupuleusement **la spécification** SystemC (Standard IEEE1666 [Ope05]), afin que l'utilisation du simulateur parallèle soit transparente pour l'utilisateur, en dehors des gains de vitesse. Nous voulons éviter toute modification ou annotation du modèle simulé.

La deuxième contrainte fondamentale est le **passage à l'échelle**. En effet, la vitesse de simulation est d'autant plus importante que les modèles sont gros. Notre outil doit donc fonctionner sur les plus gros modèles qui sont actuellement en usage. Le nombre de lignes de code C++ de ces modèles TLM peut aller jusqu'à 500 000. Toute analyse statique ayant un coût quadratique ou supérieur est donc à proscrire.

9.1.1 Le modèle d'exécution actuel de l'OSCI

La spécification de l'ordonnanceur SystemC a déjà été expliquée en détail à la section 4.1. Nous ne rappelons ici que les points significatifs dans le contexte présent.

Le comportement d'un modèle SystemC est décrit par un ensemble de processus, eux-mêmes décrits par du code C++. Pour simuler un modèle, la spécification explique comment *séquentialiser* les transitions de ces processus pour les exécuter à tour de rôle sur le processeur du simulateur. Ces explications sont données sous la forme d'un pseudo-code assez détaillé, ici schématisé par l'automate de la figure 9.1. A partir de là, l'implantation d'un *ordonnanceur* pour machine monoprocesseur est assez direct.

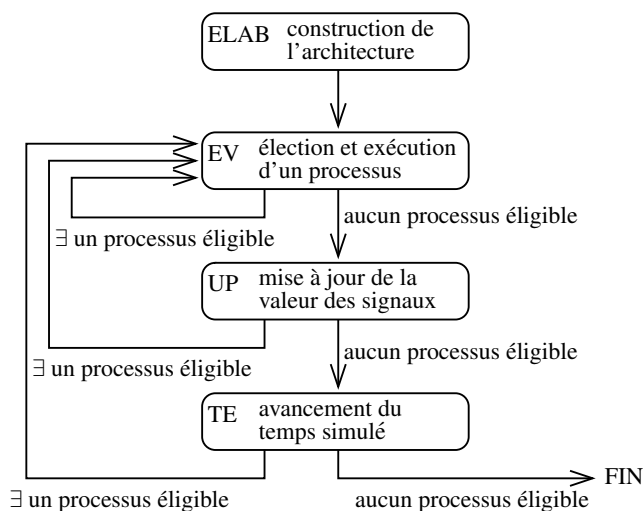


FIG. 9.1 – Automate de l'ordonnanceur SystemC.

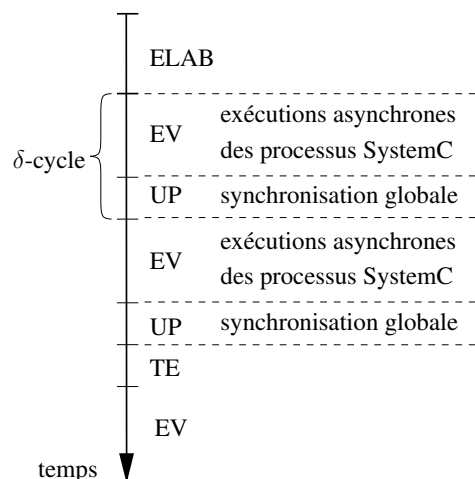


FIG. 9.2 – Diagramme d'une exécution.

Toujours selon la spécification, une simulation SystemC se découpe en plusieurs phases d'évaluations, séparées par des synchronisations globales pendant lesquelles la valeur de certains canaux de communication peut être mise à jour (figure 9.2). Durant les phases d'évaluations, l'exécution des processus est asynchrone : un processus peut être exécuté plusieurs fois alors qu'un autre attend d'être choisi par l'ordonnanceur. Par contre, une synchronisation globale ne peut avoir lieu que si plus aucun processus n'est éligible. Cela implique que seules les transitions d'un même δ -cycle sont

candidates pour une exécution en parallèle sur des processeurs séparés.

9.1.2 Parallélisation : aperçu général

La figure 9.3 donne un aperçu général du problème. Nous avons d'un côté un ensemble de *processus SystemC*, éventuellement regroupés en *modules*, et de l'autre un ensemble de *processeurs* pouvant servir à les exécuter. L'accès aux processeurs se fait via le système d'exploitation (OS). Nous créons donc un certain nombre de *processus du système d'exploitation* pour servir d'intermédiaire. Le rôle d'un ordonnanceur SystemC parallèle est d'assigner à chaque processus SystemC éligible un processus du système d'exploitation qui se chargera de son exécution. De façon analogue, le rôle de l'ordonnanceur du système d'exploitation est d'assigner à chacun de ses processus OS un processeur qui l'exécutera.

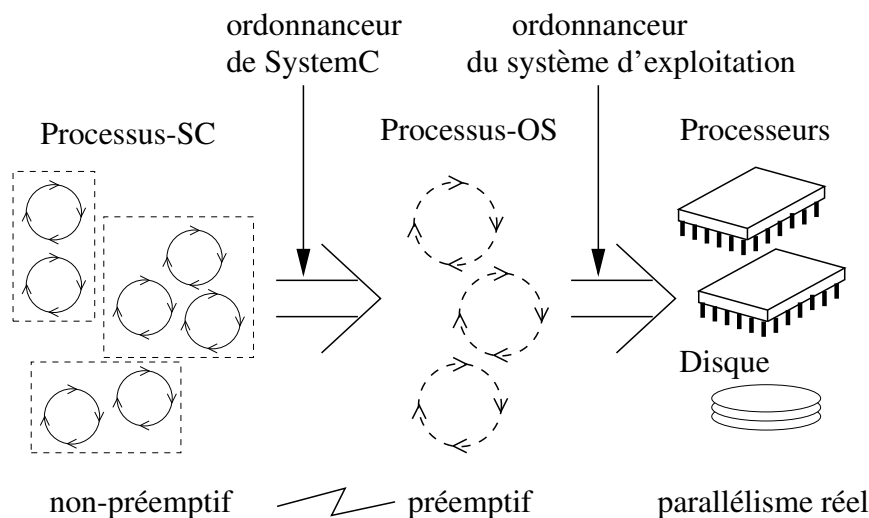


FIG. 9.3 – Des processus SystemC aux processeurs.

Le nombre de processus SystemC est fixé : autour de 50 pour les programmes qui nous intéressent. Le nombre de processeurs est en général beaucoup plus faible : souvent 2, parfois 4 et exceptionnellement 8. Le nombre de processus OS est libre. L'implantation OSCI utilise un seul processus OS. Seule une petite partie des moyens matériels est donc utilisée. A l'autre extrême, il est possible de créer autant de processus OS que de processus SystemC. Cependant, avoir un trop grand nombre de processus OS entraîne un gaspillage important des ressources. Cette solution n'est donc pas efficace en pratique. Pour profiter au mieux des capacités du matériel, la meilleure solution semble de créer autant de processus OS que de processeurs. Il peut être utile d'ajouter un ou deux processus OS pour les cas où un processus OS est en attente d'un accès aux disques.

Le problème qui nous intéresse ici est la réalisation d'un ordonnanceur SystemC parallèle, c'est-à-dire la répartition des processus SystemC sur les processus de l'OS.

9.1.3 Parallélisation : respect de la spécification

Pour respecter la spécification, la principale difficulté vient de la différence de politique entre l'ordonnanceur SystemC et l'ordonnanceur du système d'exploitation. Celui de SystemC est *non-préemptif* alors que celui de l'OS est *préemptif*. Quand l'ordonnement est *non-préemptif* (ou *collaboratif*), le processus en cours d'exécution choisit lui-même quand rendre la main à l'ordonnanceur.

Dans le cas *préemptif*, l'ordonnanceur peut interrompre un processus à n'importe quel moment, sauf à l'intérieur des sections atomiques, qui en général sont très courtes.

Il est couramment admis qu'il est plus simple de programmer des systèmes concurrents avec un ordonnanceur non-préemptif, parce que les interactions entre processus sont plus simples à prévoir et comprendre. Cela est l'une des justifications du choix fait par les concepteurs de SystemC. Par contre, si un processus ne rend jamais la main, tout le reste du système se retrouve bloqué. Utiliser un ordonnanceur préemptif résout ce problème ; et résoudre ce problème est indispensable dans le contexte d'un système d'exploitation.

Le paragraphe ci-dessous est issu de la spécification SystemC. Celle-ci autorise l'utilisation de plusieurs processeurs par une implantation SystemC mais impose que les exécutions continuent de respecter la sémantique non-préemptive (*co-routine* = processus non-préemptif).

*An implementation running on a machine that provides hardware support for concurrent processes may permit **two or more processes to run concurrently**, provided that **the behavior appears identical** to the co-routine semantics defined in this subclause. In other words, the implementation would be obliged to **analyze any dependencies** between processes and constrain their execution to match the co-routine semantics.*

Il est dit que le comportement doit paraître *identique*. Cela peut être défini de plusieurs façons. Comparer les états globaux n'est pas une bonne solution. D'une part, comparer l'état global d'une exécution parallèle avec l'état global d'une exécution séquentielle n'a de sens que si au plus un processus est en cours d'exécution ; pour de gros systèmes, la comparaison ne peut donc être possible que lors des synchronisations globales. D'autre part, comparer des états globaux est en soi un problème techniquement très difficile dans le contexte de programmes codés en C++. Il est plus simple de comparer l'accessibilité des instructions : toute instruction accessible avec une implantation parallèle doit aussi être accessible avec une implantation séquentielle valide. Pour être rigoureux, il faut imposer que lors de son exécution, les accès mémoires effectués retournent les mêmes valeurs, quel que soit le type d'implantation. La figure 9.4 propose une variante basée sur les états locaux. La comparaison formelle des différentes définitions sort du cadre de cette étude.

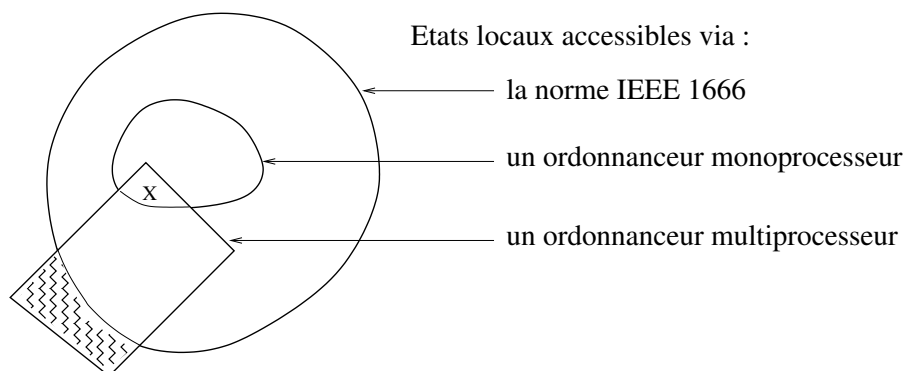


FIG. 9.4 – La croix désigne un état local. Cet état local est correct vis à vis de la spécification s'il existe un ordonnanceur monoprocesseur menant à ce même état. Un ordonnanceur multiprocesseur est correct si la partie hachurée est vide.

La spécification précise enfin que le respect de la sémantique non-préemptive va impliquer l'analyse des *dépendances* entre les processus SystemC. La formalisation et l'explication de ce point font l'objet de la section suivante.

9.2 Cadre formel : dépendances pour la parallélisation

Dans les chapitres précédents, nous avons profité de l'indépendance entre transitions pour réduire le nombre de simulations nécessaires pour une validation complète. Ici, nous allons profiter de l'indépendance entre transitions pour les exécuter en parallèle. Nous verrons que la définition de l'indépendance diffère légèrement.

9.2.1 Définitions : transitions, actions et exécutions parallèles

Nous définissons ci-dessous les structures et opérateurs nécessaires pour la suite du chapitre. Le terme "transition (SystemC)" a le même sens qu'aux chapitres précédents.

Nous appelons *transition SystemC*, ou simplement *transition*, une section de code atomique pour l'ordonnanceur SystemC. Autrement dit, il s'agit de l'ensemble du code exécuté entre le moment où l'ordonnanceur élit un processus, et le moment où le processus lui rend la main (en général via une instruction `wait`). Une transition peut donc couvrir une longue section de code, réaliser de multiples accès à la mémoire et appeler des fonctions. Il est possible qu'une transition se termine à l'intérieur de l'appel d'une fonction, éventuellement dans un module SystemC distinct.

Nous appelons *action OS*, ou simplement *action*, une section de code atomique pour l'ordonnanceur de l'OS. Cela aurait aussi pu être appelé *transition OS* ou *micro-transition*. Contrairement aux transitions SystemC, les actions OS sont en général très courtes. Il s'agit le plus souvent d'une simple lecture ou écriture d'un mot (4 octets) en mémoire, ou d'un calcul élémentaire comme par exemple une addition de deux valeurs.

Nous représentons les transitions par les lettres latines a, b, c, \dots , et les actions par les lettres grecques $\alpha, \beta, \gamma, \dots$.

Selon les définitions ci-dessus, une transition a exécute une séquence d'actions $\alpha_1 \alpha_2 \dots \alpha_i$. Étant donné une deuxième transition b exécutant les actions $\beta_1 \beta_2 \dots \beta_j$ l'*exécution parallèle* de a et b , noté $a \parallel b$, est une nouvelle séquence d'actions composée de n'importe quel entrelacement des actions de a et de b , tel que la restriction aux actions $\alpha_1, \alpha_2, \dots, \alpha_i$ conserve l'ordre original, et de même pour la restriction aux actions $\beta_1, \beta_2, \dots, \beta_j$.

Ainsi, $a \parallel b$ peut représenter $\alpha_1 \alpha_2 \dots \alpha_i \beta_1 \beta_2 \dots \beta_j$ aussi bien que $\beta_1 \alpha_1 \beta_2 \alpha_2 \dots \beta_j \alpha_i$ ou encore $\beta_1 \beta_2 \dots \beta_j \alpha_1 \alpha_2 \dots \alpha_i$. Nous pouvons ensuite construire $a \parallel b \parallel c$ en entrelaçant les actions de $a \parallel b$ avec les actions $\gamma_1 \gamma_2 \dots \gamma_k$ de c , et récursivement.

Considérons les deux transitions ci-dessous :

- $a: g++; g--;$
- $b: \text{value} = g;$

La transition a est composée de deux actions : α_1 qui correspond à l'exécution de $g++$, et α_2 qui correspond à $g--$. La transition b est composée d'une seule action β correspondant à $\text{value} = g$. La figure 9.5 représente les trois entrelacements possibles pour $a \parallel b$.

9.2.2 Relation d'indépendance pour la parallélisation

La réalisation d'un ordonnanceur parallèle correct repose sur la définition suivante :

Définition 21 — Indépendance

Soient a et b deux transitions éligibles dans un état S_1 , a et b sont indépendantes si et seulement si tous les entrelacements des actions de a et b sont valides et mènent à un même état final S_2 . Dans le cas contraire, elles sont dépendantes.

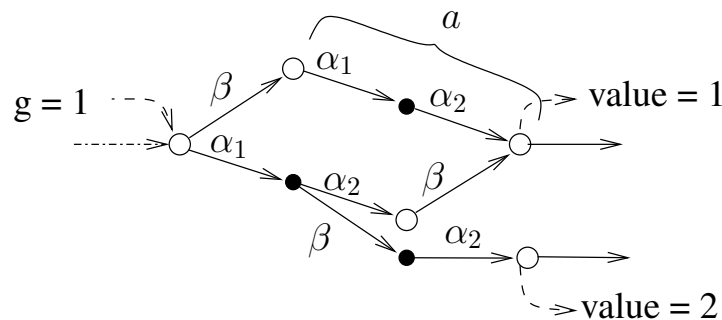


FIG. 9.5 – Entrelacements des actions de deux transitions. Les états sont représentés par des points blancs si aucune transition n’y est en cours d’exécution.

Avec les notations de cette définition, si a et b sont indépendants, alors il est correct d’exécuter $a||b$, et cette exécution mène à l’état S_2 .

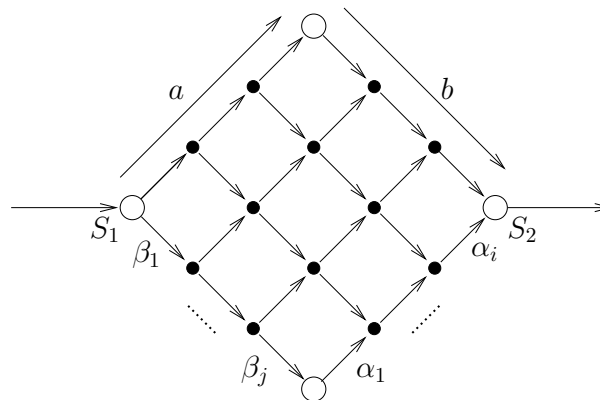


FIG. 9.6 – Transitions parallélisables.

Cette définition de l’indépendance diffère de celle du chapitre 5. En effet, dans le contexte des réductions d’ordre partiel pour la vérification, il suffit de vérifier l’existence des deux ordonnancements ab et ba et de comparer les deux états résultants ; pour la parallélisation, nous devons considérer tous les entrelacements des actions. L’indépendance pour la parallélisation implique l’indépendance pour les réductions d’ordre partiel. L’inverse est faux, comme le montre l’exemple de la figure 9.5. Cependant, la plupart des algorithmes utilisés pour évaluer l’indépendance de deux transitions à des fins de réductions d’ordre partiel, sont aussi corrects dans le cadre de la parallélisation, du fait des abstractions réalisées.

Réaliser un simulateur SystemC multiprocesseur impose de pouvoir conclure à l’indépendance de deux transitions. C’est-à-dire qu’il faut un critère capable, soit de prouver l’indépendance de deux transitions, soit de dire qu’il ne sait pas.

Si ce critère répond trop souvent qu’il ne sait pas, alors le simulateur ne va pas profiter au maximum des possibilités de parallélisation, et en conséquence des processeurs peuvent se retrouver “sans emploi”. Plus il y a de processeurs, plus il est difficile de les utiliser tous. Par contre, le nombre de transitions indépendantes croît en général avec la taille du modèle simulé, et plus précisément avec son nombre de processus SystemC. Pour tout programme SystemC, il existe un seuil théorique au delà duquel il est inutile d’ajouter des processeurs pour sa simulation. Plus un critère est puissant (c’est-à-

dire qu'il répond plus souvent "indépendant" lorsque c'est le cas), plus il est possible de s'approcher de ce seuil, mais en général un critère plus puissant demande plus de calculs et au final il peut s'avérer moins rentable. Il y a donc un compromis à trouver.

9.3 Outils existants, approche structurelle

Des simulateurs SystemC parallèles existent déjà. Les plus convaincants utilisent un *critère structurel* pour conclure à l'indépendance de deux transitions, et donc à la possibilité de les exécuter en parallèle sans changer la sémantique du programme.

Nous allons d'abord préciser le contexte d'application de ces outils, puis présenter les caractéristiques principales de deux outils opérationnels, et enfin montrer les difficultés pour étendre ces outils à notre contexte.

9.3.1 Les modèles SystemC avec communications globalement synchrones

Nous définissons ici une classe de programmes SystemC caractérisés par les canaux de communication inter-module utilisés. Cette classe de programme est très utilisée pour le développement de modèle à un niveau d'abstraction plus bas que celui des modèles transactionnels.

Un programme SystemC appartient à cette classe si et seulement si toutes les communications inter-modules se font via des canaux implantant le mécanisme de mise à jour synchrone.

Le principe de ce mécanisme est que les écritures ne sont répercutées qu'à la fin du δ -cycle (figure 9.7). Autrement dit, les lectures sont insensibles aux écritures du cycle en cours. Un corollaire est que tout processus ne peut exécuter du code que de son module originel, puisque les appels de fonctions inter-modules sont interdits. Cela permet d'éviter les dépendances à l'ordonnancement, mais comme tous les signaux sont mis à jour au même instant logique, ces canaux ne conviennent pas pour le développement de modèles (partiellement) asynchrones, comme les modèles fonctionnels.

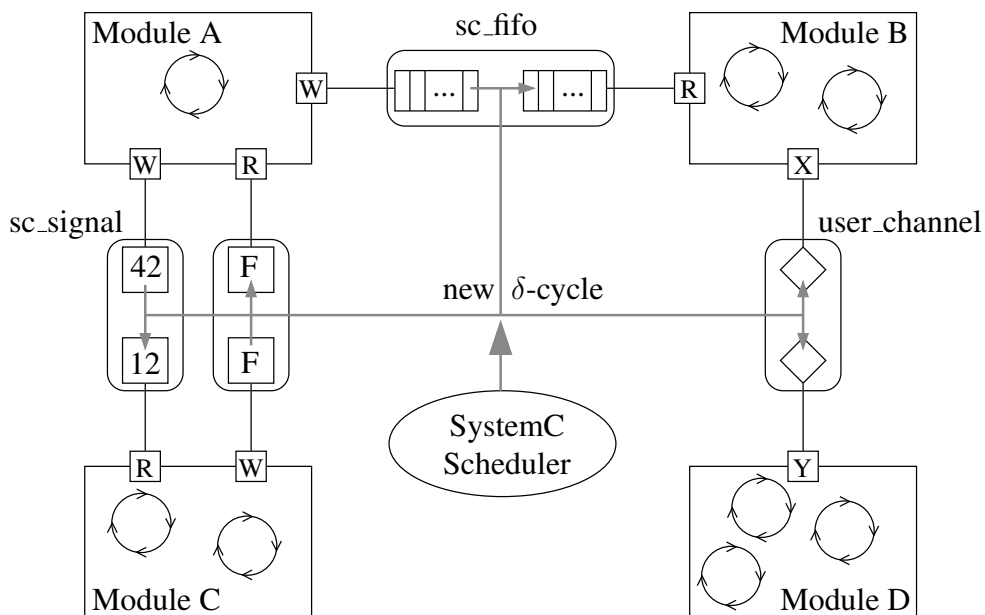


FIG. 9.7 – Modèle SystemC avec communications inter-modules globalement synchrones.

La librairie SystemC fournit deux classes de canaux primitifs utilisant le mécanisme de mise à jour synchrone : `sc_fifo` et `sc_signal` (avec ces variantes `sc_signal_rv`, `sc_signal_resolved` et `sc_buffer`). Il est possible d'en définir d'autres en héritant de la classe `sc_prim_channel` et en implantant la fonction virtuelle `update`.

La conséquence fondamentale pour le problème qui nous intéresse ici est la suivante :

Propriété 6 *Dans tout modèle n'utilisant que des canaux avec mise à jour synchrone pour les communications inter-module : si deux transitions appartiennent à des modules différents, alors elles sont indépendantes.*

9.3.2 Outils existants

Deux outils, dont la correction est fondée sur cette propriété, ont été développés. Le premier a été développé par Philippe Combes, durant sa thèse à l'université de Genève en collaboration avec STMicroelectronics [CCZ06]. Le second a été développé par Éric Paire au sein de l'équipe SPG de STMicroelectronics à Crolles [DP05].

Il y a eu d'autres travaux sur la simulation distribuée de programmes SystemC :

- Le simulateur industriel Simcluster, à priori dédié aux descriptions niveau RTL, et pour lequel peu d'informations techniques sont disponibles [Ave05].
- L'outil RITSim, décrit dans une thèse de master [Cox05].

9.3.2.1 Philippe Combes : simulation distribuée avec annotations structurelles

Le problème traité par Philippe Combes est plus large que le notre. Il s'agit en effet de répartir la simulation d'un programme SystemC sur des machines distinctes connectées par Ethernet. Dans notre cas, le fait de se limiter à des processeurs partageant la même mémoire simplifie les synchronisations entre processus et accélère les communications. L'intérêt de la simulation sur machines distinctes est évident : le coût matériel est significativement moindre (n machines monoprocesseur coûtent moins cher qu'une machine avec n processeurs).

Le partitionnement et la répartition des processus SystemC sur les différentes machines disponibles se fait **statiquement** grâce à des annotations `SC_NODE_MODULE` ajoutées au code source par l'utilisateur. Ces annotations permettent de rassembler des modules pour former des noeuds de modules, tels que les modules d'un même noeud communiquent beaucoup entre eux, et peu avec les modules des autres noeuds. Toutes les transitions d'un même noeud, et donc d'un même module, sont exécutées en séquence par une même machine. Comme toutes les communications entre les noeuds de modules se font avec des files (`sc_fifo`) ou des signaux (`sc_signal`), la propriété 6 s'applique, et la spécification est donc respectée.

Les synchronisations globales définies dans la spécification de SystemC ne peuvent pas être implantées sur des grappes de machines en utilisant simplement l'algorithme de référence. La solution mise en œuvre dans cet outil est basé le concept de *simulations parallèles à événements discrets* (*Parallel Discrete Event Simulation*, PDES), et plus précisément sur la version *conservative* (dite aussi *pessimiste*) [Bry77, CM79]. Avec cette technique, chaque machine dispose d'une horloge locale, et une machine ne fait avancer le temps localement que si elle est sûre de ne pas recevoir un message avec une date dans le passé.

Les études de cas réalisées montrent l'influence du ratio *calcul sur communication*. Plus celui-ci est élevé, plus l'accélération obtenue est grande. Ce ratio dépend a priori du type d'application, mais aussi de la pertinence des annotations du programmeur.

9.3.2.2 Éric Paire : simulation sur machines SMP avec partitionnement dynamique

La solution d'Éric Paire cible les machines multiprocesseurs. Son principal avantage est d'être entièrement automatisé. L'idée est de répartir automatiquement et **dynamiquement** les modules SystemC sur les différents processus OS. Lorsqu'un module est alloué à un processus OS, celui exécute séquentiellement les processus éligibles SystemC de ce module. Une fois qu'il n'y a plus de module à allouer, c'est-à-dire contenant des processus SystemC éligibles, la mise à jour des signaux est effectuée. Les phases de mise à jour des signaux ne sont pas parallélisées, car elles sont très courtes par rapport aux phases d'évaluation pour les modèles considérés. Cette solution est correcte vis-à-vis de la spécification tant que le programme exécuté reste dans le cadre de la propriété 6.

L'efficacité de l'outil dépend du programme exécuté. Avoir de nombreux modules améliore les résultats ; les changements de cycle très fréquents sont par contre nuisibles.

9.3.3 Relâchement de la non-préemptivité dans le cas des transactions

Les modèles transactionnels communiquent via des appels fonctionnels du composant source au composant cible. En conséquence, ils ne respectent pas les hypothèses de la propriété 6. Éric Paire a complété son outil avec un mécanisme de migration de processus SystemC entre module, dans le but de le rendre utilisable avec des communications par transactions. Cette extension du domaine d'application se fait au détriment de la non-préemptivité.

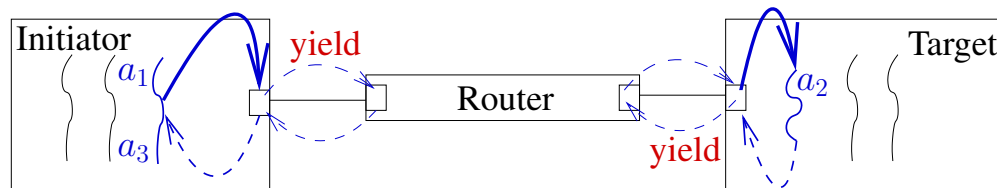


FIG. 9.8 – Réalisation d'une transaction ; insertion de points de préemption.

Considérons pour illustration la transaction schématisée par la figure 9.8. La transition a effectue une transaction vers un autre composant, par exemple en exécutant `write(addr_target, value)`. En principe, toute la transaction devrait être atomique, mais avec le mécanisme utilisé dans cette extension, la transaction est interrompue deux fois : lors de l'appel et lors de la réponse. Nous observons donc trois transitions a_1 , a_2 et a_3 au lieu d'une. La transition a_2 est généralement exécutée par un autre processus OS que ses soeurs a_1 et a_3 , puisque située dans un module différent.

Cela ajoute des comportements qui n'auraient pas été possible avec un ordonnanceur monoprocesseur valide. Cela est problématique car il arrive que la correction d'un modèle fonctionnel soit basée sur l'atomicité d'une séquence de transactions, contrairement aux modèles pour l'analyse de propriétés temporelles dans lesquels les bus ordonnent systématiquement les transactions.

La figure 9.9 montre un exemple de programme correct, mais dont l'exécution avec ce simulateur multiprocesseurs étendu peut mener à des comportements incorrects. Avec un ordonnanceur monoprocesseur, deux résultats sont possibles : "14h59" et "15h00". Avec cet ordonnanceur multiprocesseur, deux autres résultats, non cohérents, sont possibles : "15h59" (si les lectures sont exécutées entre les deux écritures) et "14h00" (si les écritures sont exécutées entre les deux lectures). Cela viole les critères de respect de la spécification SystemC, tels que présentés à la section 9.1.3.

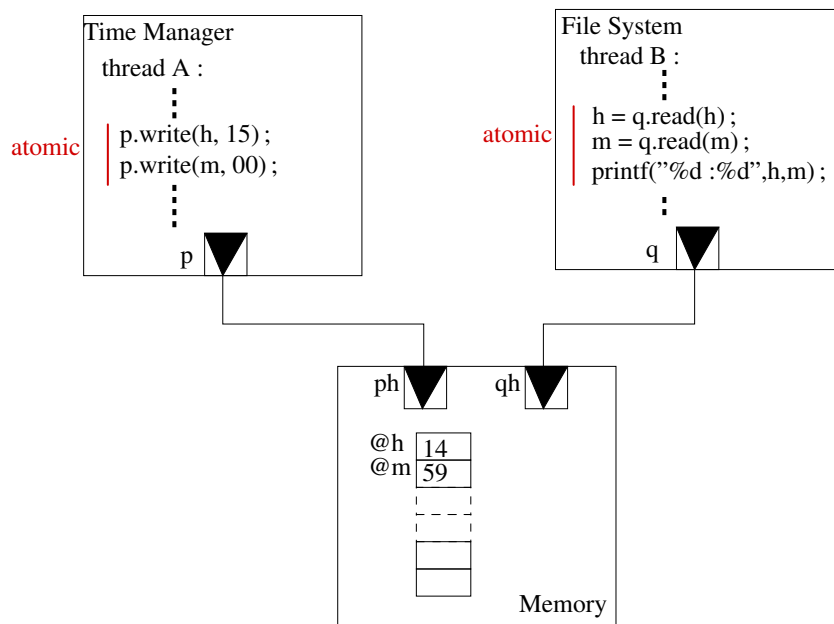


FIG. 9.9 – Extrait de modèle TLM fonctionnel.

9.4 Vers un outil adapté au TLM, approche non-structurale

L'approche structurale, qui consiste à déduire l'indépendance des transitions de la structure du modèle, a deux défauts inhérents :

- l'exploitation du parallélisme n'est pas optimale ;
- elle n'est applicable aux modèles TLM, sauf en renonçant à la non-préemptivité telle qu'imposée par la spécification.

Les approches non-structurelles, utilisant des critères plus précis, permettent de résoudre ces deux problèmes, d'une part en détectant des transactions parallélisables au sein des modules, d'autre part en tenant compte des transactions lors de l'évaluation des dépendances.

9.4.1 Les granularités envisageables

La figure 9.10 donne un aperçu des trois possibilités principales pour le niveau d'observation, qui servira à l'évaluation des dépendances.

Le tout premier schéma correspond à l'approche structurale dont nous avons déjà discuté. Pour déduire l'indépendance de deux transitions, nous ne disposons que des informations sur la structure du modèle. L'utilisation de transactions atomiques revient à avoir des variables partagées entre module : un processus ne peut pas directement modifier une variable d'un autre composant, mais il peut appeler une fonction qui la modifiera. Dans ce mini-modèle, les deux modules sont reliés par une flèche pointillée. Par conséquent, aucune parallélisation n'est possible avec l'approche structurale.

Afin de découvrir d'autres parallélisations valides, il est nécessaire d'observer le modèle de plus près. Le schéma intermédiaire de la figure 9.10 correspond à une vue au niveau processus. L'idée est ici de réaliser une analyse statique afin de déterminer pour chaque paire de processus s'ils sont susceptibles de communiquer ou d'interagir ensemble (hors canaux avec mise à jour synchrone). Cela donne un graphe, non transitif, dont les noeuds correspondent aux processus, avec une arête entre deux noeuds si les processus correspondants p et q peuvent avoir des transitions p_i et q_j dépendantes. Étant

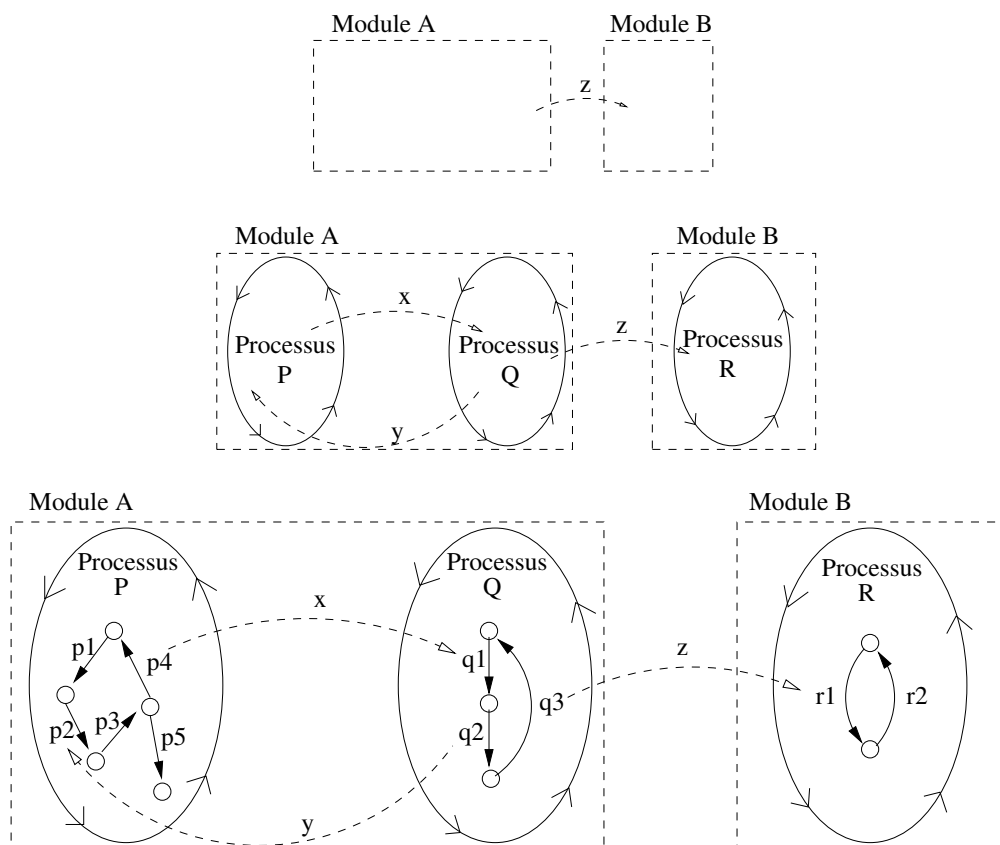


FIG. 9.10 – Représentations schématiques d'un modèle, à différents niveaux de zoom. Les rectangles pointillés représentent les modules SystemC, les grandes ellipses représentent les processus SystemC et les flèches des automates représentent les transitions SystemC. Les dépendances entre transitions sont représentées par les flèches pointillées, étiquetées par le nom des objets partagés.

donné deux transitions éligibles, il est valide de les exécuter en parallèle si les processus correspondant ne sont pas reliés directement dans le graphe. Dans le mini-modèle présenté, les processus P et R ne sont pas reliés ; nous pouvons par conséquent exécuter en parallèle des transitions de P avec des transitions de Q. Ces graphes sont semblables aux *graphes de dépendances statiques* présentés à la section 5.3.3, et que nous savons déjà partiellement construire automatiquement (section 6.6.1).

Observer le modèle au niveau des processus est plus précis que de l’observer au niveau des modules, mais il est possible, et souhaité, d’être encore plus précis. Certains composants, à l’exemple du décodeur LCMPEG (cf section 7.3), changent de phases au cours de l’exécution. Il y a des phases où ils communiquent, pour programmer un traitement ou être programmé, suivi par des phases où ils réalisent les traitements programmés. Les phases de traitement demandent souvent une grande puissance de calcul, et nécessitent peu ou pas de communications ; elles sont donc particulièrement intéressantes pour la parallélisation. Distinguer les phases de communications des phases de traitement oblige à observer l’intérieur des processus.

Le grand et dernier schéma de la figure 9.10 représente justement l’intérieur des processus sous forme d’automates dont les transitions sont des transitions SystemC. Nous supposons ici que les appels de fonctions ont été intégrés statiquement, comme des macros (*inlining* en anglais). Les différents automates sont reliés par des flèches pointillées indiquant les dépendances entre transitions SystemC. Cela donne un nouveau graphe dont les noeuds sont des transitions SystemC, et dont les arêtes indiquent leur dépendance. Désormais, deux transitions éligibles au même instant peuvent être exécutées en parallèle si elles ne sont pas reliées directement par une flèche pointillée dans ce nouveau graphe. Il est désormais possible de paralléliser des transitions de P avec des transitions de Q, par exemple p_4 et q_2 . Il s’agit toujours d’une approximation conservatrice : si deux transitions accèdent à un tableau et que l’un des index est inconnu statiquement, alors les deux transitions sont considérées dépendantes.

En résumé, plus l’observation se fait avec un grain fin, plus la simulation peut être parallélisée. La contrepartie est une analyse statique beaucoup plus complexe. Nous allons étudier plus en détail la réalisation d’un ordonnanceur multiprocesseur avec observation au niveau transitions SystemC.

9.4.2 Sketch d’ordonnanceur multiprocesseur

Voici dans les grandes lignes ce à quoi devrait ressembler un ordonnanceur multiprocesseur. Supposons que l’on connaisse dynamiquement les deux ensembles et la fonction ci-dessous :

- R : ensemble des transitions SystemC en cours d’exécution ;
- E : ensemble des transitions SystemC éligibles ;
- $L : a \mapsto L(a)$: ensemble des objets partagés auxquels la transition SystemC a peut accéder.

Chaque processus OS exécute le code suivant :

```

loop
  select  $e \in E$ 
  if  $(\bigcup_{r \in R} L(r)) \cap L(e) = \emptyset$ 
  then run  $e$ 

```

L’instruction “run e ” déplace la transition e de l’ensemble E à l’ensemble R , puis donne la main au processus SystemC correspondant. Les notifications ont pour effet d’ajouter des transitions dans l’ensemble E . Pour être complet, il faudrait ajouter le code pour le traitement des phases de synchronisations globales (changement de cycle).

Mettre en œuvre cet algorithme n’est pas simple car il faut prévoir des verrous suffisamment nombreux pour que le fonctionnement soit correct, mais sans que cela dégrade de trop les performances,

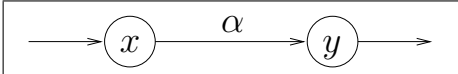
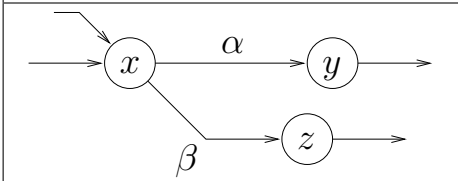
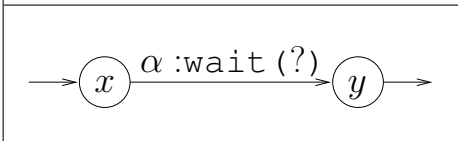
ou puisse causer un interblocage. Idéalement, il faudrait qu'un ordonnanceur multiprocesseur utilisé sur une machine monoprocesseur soit aussi efficace que l'ordonnanceur monoprocesseur de référence. Il faut donc limiter les calculs dynamiques, mais sans oublier qu'un processus OS sans transition SystemC à exécuter constitue un gaspillage de ressource matériel.

L'autre difficulté consiste à calculer $L(a)$, ce qui se fait par une analyse statique préalable.

9.4.3 Analyses statiques pour le pré-calcul des dépendances

Étant donné un point du code x d'un processus SystemC, nous devons déterminer l'ensemble $L(x)$ des objets accessibles lors d'une exécution, sans franchir une instruction `wait` qui marquerait la fin de la transition. Vu le contexte, il est indispensable que cette analyse passe à l'échelle. Son coût doit donc être linéaire, ou éventuellement d'ordre $n \log(n)$ avec n caractérisant la taille du programme à exécuter. Le plus simple pour définir cette analyse est de se baser sur le *flot de contrôle*. Comme pour l'algorithme de l'ordonnanceur, nous allons rester dans les grandes lignes, afin de donner une idée de ce qu'il faudrait faire et de quelles sont les principales difficultés.

Pour chaque instruction α , nous notons $\lambda(\alpha)$ l'ensemble des objets partagés touchés par cette instruction. Par exemple : $\lambda(T[i] = 42) = \{i, T\}$ et $\lambda(\text{wait}(\text{event})) = \{\text{event}\}$.

	$L(x) = \lambda(\alpha) \cup L(y)$
	$L(x) = \lambda(\alpha) \cup L(y) \cup \lambda(\beta) \cup L(z)$ exemple : $L(\text{if } (1+2-3) \text{ then } x++ \text{ else } y++) = \{x, y\}$
	$L(x) = \lambda(\alpha)$ exemple : $L(\text{wait}(10, \text{SC_NS}) ; x++) = \emptyset$

TAB. 9.1 – Cas élémentaires pour le calcul de $L(x)$.

Une fois le flot de contrôle disponible, chaque noeud x est annoté par $L(x)$, en fonction des règles décrites par la table 9.1. Chaque noeud n'est traité qu'une seule fois. Essayer d'évaluer les conditions est en général trop coûteux, mais des extensions à la version de base peuvent être envisagées pour certains cas particuliers.

Plutôt que de complexifier l'analyse statique pour les cas où les abstractions réalisées sont trop fortes, il peut être préférable de paramétrer l'ensemble $L(x)$ par des informations dynamiques. Par exemple, si nous savons que la valeur de l'index i sera connue au début de la transition, nous pouvons placer l'élément $T[i]$ dans $L(x)$ plutôt que le tableau T entier. De même, si une transition commence par un branchement de condition c vers deux noeuds fils x_c et x_{-c} , nous pouvons dans certains cas évaluer c dynamiquement afin de choisir entre $L(x_c)$ et $L(x_{-c})$.

Il existe un autre cas qui mérite une attention particulière. Dans un modèle TLM, nous n'écrivons pas `timer.start()` mais `port.write(timer_addr+start_offset, 1)`. Une analyse naïve mènerait à la conclusion que chaque transition initiant au moins une transaction est dépendante avec toutes les transitions liées à un port cible. Il serait donc très rentable de tenir compte du paramètre adresse des transactions, qui dans une grande majorité des cas est soit constant, soit aisé à borner. Un

tel traitement nécessite bien sûr de connaître la *carte des adresses mémoires* (*memory map*, dans le jargon). A noter que cela serait utile quel que soit le grain d'observation choisi.

9.4.4 Identification dynamique des transitions

Lors d'une exécution, il est facile d'obtenir les identifiants des processus éligibles. Il est beaucoup plus difficile de savoir à quelle ligne du code se situe un processus, or cela est nécessaire afin d'utiliser les résultats de l'analyse statique.

Dynamiquement, la position dans le code est déterminée par un pointeur de code en mémoire et les adresses de retour présentes dans la pile d'exécution. Les informations de debug permettent en principe de faire le lien avec le code source et donc les résultats de l'analyse statique, mais il faudrait une librairie pour pouvoir les exploiter.

Une autre solution serait d'instrumenter le code lors de l'analyse statique avec des instructions précisant l'identifiant de la transition à venir : `wait(X)` deviendrait par exemple `I_am_here(transition_id) ; wait(X)`. Cette approche risque de soulever les mêmes difficultés que celle déjà suivie pour la validation (cf 6.3.3).

9.5 Perspectives

Nous avons d'abord défini ce qu'est une parallélisation valide. Nous avons ensuite expliqué le fonctionnement de deux outils existants ainsi que leurs limitations pour le contexte des modèles fonctionnels. Enfin, nous avons donné des pistes pour réaliser un nouvel ordonnanceur multiprocesseur plus adapté et plus puissant. Cependant, nous sommes encore loin d'aboutir à un outil complet et fonctionnel. Les difficultés restantes se répartissent en trois classes :

- l'analyse statique ;
- l'implantation efficace de l'ordonnanceur ;
- le lien entre les informations statiques et dynamiques.

Pour le premier point, il s'agit essentiellement de problèmes théoriques. Les deux autres sont plus techniques ; ils requièrent des connaissances poussées en programmation avec processus OS, et en compilation.

Vu le travail qui semble encore nécessaire, il serait intéressant d'arriver à estimer l'accélération que nous pourrions obtenir sur des études de cas réelles, avant d'avoir implanté un prototype complet. De plus, il faut aussi regarder si une partie du travail à accomplir pourrait servir pour d'autres sujets de recherche. Disposer d'une librairie pour accéder directement aux informations de debug sans passer par un débogueur comme `gdb` permettrait sans doute d'améliorer et d'accélérer l'enregistrement des traces détaillées (section 6.6.2). Les analyses statiques pourraient aussi servir à d'autres outils, par exemple pour la vérification par modèle, puisque les réductions d'ordre partiel statiques nécessitent aussi des analyses préalables des dépendances. L'outil `LusSy` [MMMC06] construit d'ores et déjà la structure de contrôle, et réalise des abstractions spécifiques pour les adresses. Un autre doctorant, travaillant à STMicroelectronics, réfléchit actuellement au problème inverse : c'est-à-dire à la génération de code SystemC à partir d'automates avec macro-transitions, qui contiennent justement les informations nécessaires pour nos analyses statiques.

Chapitre 10

Conclusion et perspectives

Sommaire

10.1 Bilan	153
10.1.1 Mise en évidence du problème de l'ordonnancement	154
10.1.2 Détection des erreurs de synchronisation	154
10.1.3 Extension aux modèles avec temps imprécis	155
10.1.4 Autres contributions	155
10.2 Perspectives	156
10.2.1 Amélioration des performances	156
10.2.2 Réduction du nombre d'entrelacements visités pour LUSY	158
10.2.3 Extensions vers d'autres espaces de données ou d'autres contextes	159

10.1 Bilan

L'équipe SPG de STMicroelectronics tient une place prédominante dans le développement des modèles transactionnels dans l'industrie. Ces modèles correspondent à un nouveau niveau d'abstraction, qui repose sur une façon simplifiée et désormais normalisée de communiquer.

Dans un programme SystemC au niveau RTL, les canaux de communications servent à la fois pour les communications entre modules et pour les synchronisations entre processus. Ces canaux sont implantés de telle sorte que l'ordonnancement n'ait pas d'effet sur la fonctionnalité.

En revanche, pour les modèles transactionnels, les synchronisations entre processus SystemC ne sont plus aussi directement liées aux communications inter-composants. Les processus peuvent exécuter du code d'autres modules via les transactions, qui en pratique sont implantées par des appels de fonctions atomiques. Une conséquence est que le comportement d'un modèle SystemC-TLM peut dépendre de l'ordonnancement. L'inconvénient est qu'une simulation avec un ordonnancement arbitraire peut cacher des erreurs de synchronisation. L'avantage est que cela permet de modéliser un sur-ensemble des comportements de la puce finale, plutôt qu'un sous-ensemble. Cela n'avait, jusqu'au début de la collaboration ST-Verimag, pas fait l'objet de travaux poussés, d'où un empirisme certain dans les synchronisations des modèles industriels développés.

Le premier docteur issu de cette collaboration, Matthieu Moy, a proposé une solution visant à prouver formellement et automatiquement que les synchronisations d'un modèle sont exemptes d'er-

reurs. La chaîne d'outils LUSSY, qu'il a développé, permet d'ores et déjà de prouver des propriétés sur des modèles de petite taille.

Bien sûr, la vérification formelle ne peut venir, seule, à bout de la complexité croissante des modèles de systèmes sur puce d'aujourd'hui et de demain. D'autres pistes devaient donc être explorées, la première idée étant d'avoir recours à la simulation. Durant la thèse décrite dans ce document, nous avons poursuivi et avancé sur l'identification, la résolution puis l'extension du problème de l'ordonnancement en SystemC-TLM.

10.1.1 Mise en évidence du problème de l'ordonnancement

Les développeurs de modèles transactionnels viennent, pour la plupart, du monde de la description du matériel et donc des modèles synchrones. L'absence d'horloges, la programmation asynchrone et les dépendances à l'ordonnancement constituaient donc souvent des problématiques mal connues. La première tâche, entamée par Matthieu Moy, a donc été de clarifier le fonctionnement de l'ordonnancement en SystemC, et son impact sur la sémantique des modèles TLM.

Lorsqu'une modèle fonctionne correctement avec l'ordonnancement par défaut de l'implantation OSCI, nous avons montré que la moindre modification dans les sources ou les données pouvait complètement modifier cet ordonnancement par défaut. La fiabilité d'un modèle doit donc passer par un contrôle des dépendances à l'ordonnancement. Cela n'était pas acquis dans tous les milieux.

Nous avons ensuite isolé les exemples de mauvaises synchronisations les plus fréquentes, puis nous avons proposé des structures permettant de résoudre ces problèmes : les événements persistants (`pevent`) et les verrous avec priorités (`tlm_arbiter`).

Ces deux structures et quelques conseils ne suffisent bien sûr pas à éviter toutes les erreurs de synchronisations. Par ailleurs, l'indéterminisme de l'ordonnancement peut aussi servir à modéliser des informations encore inconnues au niveau RTL. De ces constats, nous avons montré qu'explorer l'espace des ordonnancements, en plus de celui des données, est une nécessité pour développer des modèles fiables et représentatifs du matériel.

10.1.2 Détection des erreurs de synchronisation

Le problème des dépendances à l'ordonnancement, dans le cadre de SystemC, a aussi été relevé par d'autres équipes de recherche ou de développeurs d'outils. Au mieux, des ordonnanceurs aléatoires ont été proposés. Cette solution était malheureusement bien trop simpliste et peu efficace par rapport au problème soulevé.

Il s'agit, en revanche, d'un problème bien connu dans d'autres domaines, par exemple ceux touchant au logiciel concurrent. Les techniques de *réduction d'ordre partiel* et de *vérification à la volée* y sont utilisées depuis longtemps, et une nouvelle technique, la *réduction d'ordre partiel dynamique*, a été récemment publiée [FG05].

Nous avons adapté puis implanté la réduction d'ordre partiel dynamique pour les modèles SystemC-TLM. La chaîne d'outils développée permet d'explorer, avec une *couverture totale*, n'importe quel programme codé en SystemC, et ce pour des tailles bien supérieures à ce que les techniques précédentes permettaient. Grâce à notre analyseur et aux outils périphériques, nous avons mis en évidence une erreur de synchronisation non-triviale dans un modèle industriel. Ces travaux ont en partie été publiés dans [HMMCM06] (le nouvel algorithme et la nouvelle preuve n'ont été formalisés qu'après).

Il s'agit du premier outil de ce type dans le domaine des modèles transactionnels. D'un point de vue théorique et plus global, les contributions sont plus restreintes. La principale est d'avoir proposé

un nouvelle boucle pour l'algorithme principal de génération, dont la preuve est basée sur des *arbres de contraintes d'ordonnements*. Ce nouvel algorithme est plus simple à comprendre et démontrer, et devrait donc être une très bonne base pour les évolutions futures.

Par *couverture totale* de l'espace des ordonnancements, nous signifions que pour un jeu de données fixées (et des durées fixées), nous détectons toutes les erreurs locales à un processus, ainsi que tous les inter-blocages possibles. Ce genre de garantie n'est en général fourni que par les outils de vérification formelle.

Une grande majorité des chercheurs en vérification formelle travaillent sur des langages spécifiques, possédant un nombre très limité de structures élémentaires, ainsi qu'une sémantique formalisée. Comme les études de cas industrielles ne sont généralement pas écrites dans ces langages, une traduction est nécessaire. Si elle est faite à la main, des erreurs peuvent malencontreusement être introduites. Automatiser cette traduction est déjà en soi un problème complexe, car il faut tenir compte des nombreuses structures du langage d'origine, ainsi des ambiguïtés et indéterminismes de sa sémantique. Le résultat est toujours un modèle abstrait par rapport à l'original. Ces abstractions facilitent la preuve de propriétés vraies, mais rendent incomplète la génération de contre-exemples effectifs pour les propriétés fausses. Dans cette thèse nous avons choisi de travailler directement sur le code source C++. En conséquence, les erreurs détectées sont de vraies erreurs, et nous savons les relier au code source, ce qui est primordial pour leur correction.

10.1.3 Extension aux modèles avec temps imprécis

D'autres travaux, réalisés aussi dans le cadre de cette collaboration ST-Verimag, ont montré l'importance de relâcher les informations temporelles des modèles fonctionnels, afin de représenter, non plus un sous-ensemble, mais un sur-ensemble des comportements réalistes. Des programmes SystemC avec délais bornés sont ainsi apparus. Comme pour l'indéterminisme de l'ordonnement, la première idée mise en œuvre fut un générateur aléatoire. De nouveau, le problème est que cela n'apporte aucune garantie sur l'absence d'erreurs.

Nous avons alors décidé d'étendre notre chaîne d'outils pour couvrir efficacement l'espace des jeux de durées valides. Cette fois, nous avons développé à la fois une nouvelle technique théorique, et un nouvel outil. Comme pour le problème de l'ordonnement, nous travaillons directement sur le code source C++ exempt d'abstraction, et nous fournissons malgré cela une garantie formelle : pour un jeu de données fixé, nous détectons de nouveaux toutes les erreurs locales et tous les inter-blocages. Notre nouvelle technique étend la *réduction d'ordre partiel dynamique* avec de la *programmation linéaire*, dédiée à les résolutions des contraintes temporelles engendrées.

Sans surprise, la présence de délais bornés augmente significativement le coût de la validation par rapport au problème précédent. Malgré cela, nous sommes toujours capable de traiter des études de cas industrielles, mais nous savons qu'il y a peu de chose à espérer avec cette technique pour les modèles de grandes tailles. Pour ceux-ci, il reste nécessaire d'en isoler des sous-systèmes. A ce sujet, nous avons montré que les principales optimisations possibles ne pourront être obtenues que grâce à l'amélioration de la partie *réduction d'ordre partiel dynamique*, soit en raffinant la relation de dépendance calculée, soit en améliorant l'algorithme principal.

Cette extension a aussi fait l'objet d'une publication : [HMMC06].

10.1.4 Autres contributions

Nous avons décrit les principales contributions de cette thèse. Il en existe d'autres, mineures ou simplement moins visibles.

J'ai commencé avec une connaissance très limitée de la conception des systèmes sur puce, et peu d'expérience de la programmation concurrente. Au bout de trois ans, il en bien autrement. Grâce aux connaissances et à l'expérience accumulées, j'ai pu aider mes collègues à avancer dans leurs travaux respectifs.

Le premier exemple concerne la parallélisation du simulateur SystemC, qui a été présentée au chapitre 9. Il s'agit du sujet confié à Youssef Bouzouzou pour son DRT¹, qu'il a attaqué en 2006. Mes connaissances sur les relations de dépendances entre processus asynchrones ont permis de définir rapidement un cadre théorique propre, profitant des grandes similitudes entre les réductions d'ordre partiel (statiques) et le problème de la parallélisation. Nous savons désormais quels sont les problèmes à résoudre pour obtenir un simulateur SystemC pour modèles TLM à la fois efficace et fiable.

Je pense aussi avoir aidé les collègues lors de discussions plus informelles et (réciproquement) bénéfiques. Au deuxième chapitre, j'ai fait mention des travaux de Jérôme Cornet sur le raffinement de modèles fonctionnels en modèles temporisés. Au moment où j'écris cette conclusion, il travaille à montrer que ce raffinement est *correct par construction* (moyennant certaines règles), via une modélisation formelle basée sur des automates *MICmac* munis d'un produit spécifique au parallélisme SystemC. Là aussi, mais dans une moindre mesure, j'ai pu donner quelques conseils pertinents (malheureusement parfois sous la forme de contre-exemples pour des versions intermédiaires, mais qu'il a toujours rapidement corrigées). Ayant travaillé aussi avec les ingénieurs de l'équipe SPG de STMicroelectronics, des échanges similaires ont eu lieu.

Pour conclure, je tiens à rappeler que cette thèse s'est déroulée au contact du monde industriel, et s'est attaquée à des problèmes bien réels, qui nuisaient au travail des développeurs de modèles transactionnels de systèmes sur puce. Grâce aux travaux effectués, par moi et mes collègues, nous les avons aidés à formaliser ces problèmes, et avons développé des techniques et outils pour y faire face.

10.2 Perspectives

Il serait dommage de s'arrêter là. De nombreuses pistes, pouvant apporter une véritable aide pour les développeurs, restent à explorer. Voici ci-dessous celles qui nous semblent les plus prometteuses.

10.2.1 Amélioration des performances

Il existe tout d'abord plusieurs pistes pour accroître la complexité des programmes traitables.

10.2.1.1 Clarification des synchronisations au niveau transactionnel

En théorie, prendre uniquement en compte les variables partagées et les événements SystemC pour le calcul des dépendances, est suffisant pour traiter tous les modèles TLM. En pratique, cela revient à re-valider éternellement des structures élémentaires, pourtant connues comme étant fiables.

Sur notre principale étude de cas, nous avons réduit d'un facteur 4 le nombre d'ordonnements visités, uniquement grâce à la prise en compte des événements persistants. En comptant en nombre de dépendances permutable, nous sommes passés de 7 à 5. Parmi les 5 restantes, 3 correspondent à du polling et 2 correspondent à l'erreur découverte. Pour l'erreur découverte, la seule chose à faire est de la corriger. Pour les 3 dépendances restantes, l'ordonnement a une conséquence sur la date de réception d'un événement. Une idée possible consiste à abstraire cette conséquence temporelle avec une instruction `PV_wait`. Pour cela, il suffit de remplacer le polling `while(!b) wait(T)`

¹ *Diplôme de Recherche Technologiques*, ~ mini-thèse technique sur 2 ans.

par `wait (b_event)` ; `PV_wait ([0, T])`. Il s'agit d'une abstraction conservatrice. Cela ajoute un délai borné mais supprime une dépendance permutable. Il est probable que le gain soit positif, puisque chaque dépendance permutable évitée réduit le coût total d'un facteur 2.

Concernant l'exemple dit "TP-TLM", utiliser des événements persistants suffit à supprimer toutes les dépendances permutable, et donc à ramener à 1 le nombre d'ordonnements qu'il est nécessaire de visiter. En revanche, les événements persistants sont inutiles pour l'exemple de l'indexeur, dont la fonctionnalité est pourtant correcte quel que soit l'ordonnement. Cependant, une table d'association peut aussi être vue comme une structure de communication de haut-niveau. En se basant sur le fait que, dans une table d'association, une donnée est identifiée par sa clef et non par son adresse mémoire, nous pourrions supprimer des permutations inutiles. Cela suppose bien sûr que les fonctions d'insertion, de suppression et d'accès soient correctement codées.

Il existe d'autres structures de synchronisations de haut-niveau dont la prise en compte pourrait éviter la génération d'ordonnements redondants, par exemple la variante avec compteur des événements persistants, ou encore le système d'arbitrage robuste conçu pendant cette thèse. Actuellement, les développeurs de modèles TLM, au mieux recopient du code reconnu correct, au pire écrivent du code faux. Définir et utiliser des classes réutilisables permet de faciliter à la fois le développement et la validation des modèles.

Plus généralement, tout objet ou composant accédé par au moins 2 processus, peut être le sujet de ce type d'optimisation. Pour chaque nouvelle structure, le protocole à suivre est le suivant :

1. une nouvelle classe doit être définie si elle n'existait pas encore, afin de fixer son implantation et d'éviter les variantes erronées ;
2. il faut prouver, par un moyen ad hoc, que l'implantation garantit une propriété du type : *si les appels à l'interface suivent telles contraintes sur leur ordonnancement, alors les réponses de la structure sont fonctionnellement équivalentes* ;
3. l'analyseur est complété d'après la propriété ci-dessus, et il reste à évaluer les résultats..

10.2.1.2 Algorithmes pour la réduction d'ordre partiel

D'autres optimisations plus générales sont à chercher dans la technique de réduction d'ordre partiel. Nous avons mis en évidence des cas où notre algorithme génère des ordonnancements qui sont pourtant équivalents d'après la relation de dépendance utilisée. Des travaux supplémentaires permettraient sans doute d'éviter certaines redondances dans l'ensemble d'ordonnements généré. Cependant, il n'est sûr que ces cas soient suffisamment fréquents dans les programmes réels pour que de telles optimisations soient rentables.

Une autre solution est de réduire ses ambitions. Actuellement, nous obtenons tous les états finaux possibles, donc nous détectons à la fois toutes les erreurs locales et tous les inter-blocages. La méthode nommée "*net unfolding*" [McM92b] évite la construction explicite des états globaux, et en conséquence détecte toutes les erreurs locales, mais pas les inter-blocages. A priori, cette technique reviendrait à générer tous les *passés* de transition, selon la définition de la sous-section 5.3.4. Il est facile de trouver des exemples où cette technique devrait permettre de réduire très significativement le coût de la validation des propriétés locales. La question est d'une part de trouver comment adapter cette méthode à des langages de haut-niveau, d'autre part de savoir si elle sera aussi efficace dans les cas réels.

Par ailleurs, nous ne tenons pas directement compte de la propriété à vérifier. Actuellement, un ensemble de propriétés locales est validé en une seule génération d'ordonnements, et le nombre de ces propriétés est sans effet sur le coût total. Si ce coût est trop important, il peut être intéressant

d'avoir un outil qui abstrait tout ce qui n'a pas de conséquence sur une propriété particulière. Il y aurait peut-être là aussi des recherches à mener pour éviter des permutations, ou des combinaisons de permutations, qui n'ont aucun impact sur cette propriété particulière. Appliquer du *slicing* [Tip95] serait déjà un début. Intuitivement, les jeux d'ordonnements ou de durées générés ne paraissent par encore optimaux.

10.2.1.3 Autres optimisations possibles

Parmi les optimisations restantes, la plus simple et la prometteuse consiste à réaliser une version parallèle de nos outils `rvs` et `rvt`. Chaque fois que l'analyseur génère plusieurs nouvelles permutations pour une même trace, cela donne des tâches parfaitement indépendantes. Chacune de ces permutations peut donc être explorée par un processus distinct, sur une machine distincte. Il s'agit de la même idée qui est mise en œuvre par la paire d'outils `make -j` et `distcc` [Poo03]. Avec une douzaine de machines de bureau, ce qu'il est facile de trouver dans une équipe de recherche (au moins la nuit), nous pouvons espérer un gain d'au moins un facteur 10.

Une autre optimisation prometteuse, et même indispensable si l'on veut se comparer à des outils comme VeriSoft [God05], est d'utiliser du *backtracking* (retour arrière) pour éviter de recommencer chaque simulation depuis l'état initial. Cela suppose d'enregistrer des états intermédiaires, et de savoir relancer des exécutions depuis ces états. Le facteur, que nous pouvons gagner avec cette amélioration, est ici une fraction de la longueur des simulations.

A la fin du chapitre 5, nous avons discuté de la vérification d'une propriété globale, via son codage sous la forme d'un oracle intégré au système sous test. Cela permet de se ramener au problème de la vérification de propriétés locales, mais augmente le nombre d'ordonnements à visiter. L'ensemble des traces analysées contient cependant toute l'information nécessaire pour vérifier une propriété globale, même sans passer par l'intégration de son oracle. Pour profiter de ces informations, il faudrait écrire, non plus un oracle pour trace d'exécution, mais un oracle pour classe d'équivalence. Concrètement, il s'agit d'écrire un oracle qui tienne compte que l'ordre de certains événements est inconnu, et qui répond soit que la propriété est vraie, soit qu'elle violable avec telle linéarisation de l'ordre partiel issu de l'analyse. Des travaux portant sur cette même question ont été publiés dans [BL03].

10.2.2 Réduction du nombre d'entrelacements visités pour LUSSY

Actuellement, les outils de vérification branchés sur la sortie de LUSSY doivent explorer un très grand nombre d'entrelacements. En effet, le codage de programmes asynchrones en programmes synchrones ne résout pas le problème issu du très grand nombre d'entrelacements possibles. Il en est de même pour les techniques de vérification symbolique. Il serait intéressant d'appliquer les enseignements de cette thèse, en particulier sur les dépendances en SystemC, aux backends de LUSSY.

Par exemple, nous avons pu estimer les différences de complexité entre les versions sans temps, avec durées bornées et avec durées fixes d'un même modèle. La validation de la version sans temps nous a paru hors d'atteinte, or c'est cette version que LUSSY adresse actuellement. En permettant à LUSSY de générer des modèles pour les autres versions, il serait certainement possible de vérifier des exemples industriels de plus grande taille, en contrepartie de garanties moindres sur la robustesse.

[KGS06] propose une solution pour combiner les réductions d'ordre partiel (statiques) avec des techniques symboliques, à la fois compatibles avec les méthodes BDD et SAT. Des contraintes sont ajoutées pour réduire le nombre d'ordonnements possibles, tout en préservant les éventuelles

erreurs. Le concept de *transactions* utilisé dans ces travaux devraient permettre de simuler la non-préemptivité de SystemC².

Bien sûr, il serait intéressant de pouvoir faire de même avec une réduction d'ordre partiel dynamique, mais cela soulève plusieurs difficultés supplémentaires.

1. Avec les techniques de réduction dynamiques, des ordonnancements sont ajoutés à un singleton initial, au lieu d'en enlever. Il faudrait donc retirer dynamiquement des contraintes à un système en cours de résolution.
2. Il faut prévoir un système pour calculer l'ordre partiel, et le faire interagir avec le système de contraintes booléennes générées.
3. La réduction d'ordre partiel dynamique n'a pas été conçue pour la validation de programmes cycliques, avec stockage et comparaison d'états.

10.2.3 Extensions vers d'autres espaces de données ou d'autres contextes

Nous avons traité deux sources d'indéterminisme : l'ordonnement et le temps mou. Les travaux présentés dans [GKS05] propose une solution analogue pour des données numériques. Il s'agit de générer automatiquement des données en fonction de traces d'exécutions détaillées, qui décrivent l'utilisation des entrées. Chaque fois qu'une entrée est utilisée dans une condition, éventuellement de façon indirecte (e.g. : `x=input(); y=x+2; if (y>12) . . .`), alors un système d'équations est construit, tel que ses solutions inversent la valeur de la condition. Si il est possible d'en extraire une solution, celle-ci sert d'entrée pour un nouveau test, qui sera exécuté puis analysé comme le précédent. Bien sûr, pour certains types d'équation, il n'existe pas de solveur capable d'évaluer l'existence d'une solution et d'en extraire une le cas échéant. Cette technique a été appliquée conjointement avec de la réduction d'ordre partiel dynamique dans [SA06]. Une intégration dans notre chaîne d'outils pour SystemC et TLM aurait certainement des utilités.

Fin 2006, Le projet OpenTLM a vu le jour, au sein du pôle de compétitivité Minalogic. Ce projet vise le développement et la validation des modèles de SoC écrits en SystemC-TLM. Il rassemble des entreprises (STM, Thomson, Silicomp et KeesDA) et des laboratoires publics (Verimag, CEA-Leti, INRIA et TIMA). La chaîne d'outils présentée dans ce document sera intégrée à ce projet. Cela va permettre de l'évaluer sur une plus grande variété d'études de cas, d'avoir des retours d'utilisateurs et de la compléter ou l'améliorer en conséquence. Les programmes développés au sein du projet OpenTLM, dont cette chaîne d'outils, seront diffusés en open-source.

Durant cette thèse, nous nous sommes concentrés sur les modèles SystemC, transactionnels et fonctionnels. Bien sûr, les techniques employées sont probablement applicables à d'autres domaines. La plupart ont justement été inspirées elles-mêmes par des techniques d'autres domaines d'application. Le monde de l'industrie regorge encore de nouvelles problématiques à étudier.

²d'après une discussion avec l'un des auteurs.

Annexe A

Classe pour un arbitrage indépendant de l'ordonnancement

Les explications correspondantes se trouvent dans la sous-section [4.3.2](#).

A.1 Entête

```
#include <set>
#include <vector>
#include <systemc.h>
using namespace std;

class tlm_arbiter : public sc_prim_channel {
public:
    tlm_arbiter(unsigned size_);
    tlm_arbiter(const char* name_, unsigned size_);
    ~tlm_arbiter();

    void lock(unsigned id); // id must be in [0..size_)
    void unlock();         // low id = high priority

    virtual const char* kind() const {
        return "tlm_arbiter";
    }

protected:
    virtual void update();
    void init(unsigned size);

    bool in_use;
    set<unsigned> new_requests;
    set<unsigned> pending_requests;
    vector<sc_event *> events;
};
```

A.2 Implantation

```

tlm_arbiter::tlm_arbiter(unsigned size_) :
    sc_prim_channel(sc_gen_unique_name("arbiter")),
    in_use(false), new_requests(), pending_requests(), events() {
    init(size_);
}

tlm_arbiter::tlm_arbiter(const char* name_, unsigned size_) :
    sc_prim_channel(name_),
    in_use(false), new_requests(), pending_requests() {}

void tlm_arbiter::init(unsigned size) {
    for (unsigned i = 0; i < size; ++i)
        events.push_back(new sc_event());
}

tlm_arbiter::~~tlm_arbiter() {
    unsigned size = events.size();
    for (unsigned i = 0; i < size; ++i)
        delete events[i];
}

void tlm_arbiter::lock(unsigned id) {
    assert(id < events.size());
    new_requests.insert(id);
    request_update();
    wait(*(events[id]));
}

void tlm_arbiter::unlock() {
    assert(in_use);
    if (!pending_requests.empty()) {
        set<unsigned>::iterator i = pending_requests.begin();
        events[*i]->notify();
        pending_requests.erase(i);
    } else {
        in_use = false;
    }
}

void tlm_arbiter::update() {
    pending_requests.insert(new_requests.begin(),
                           new_requests.end());
    new_requests.clear();
    if (!in_use) {
        set<unsigned>::iterator i = pending_requests.begin();

```

```
        events[*i]->notify_delayed();
        pending_requests.erase(i);
        in_use = true;
    }
}
```

A.3 Test

Programme de test

```
class top : public sc_module
{
public:
    tlm_arbiter arbiter;
    SC_HAS_PROCESS(top);
    top(sc_module_name name) :
        sc_module(name),
        arbiter(3) {
        SC_THREAD(R);
        SC_THREAD(Q);
        SC_THREAD(P);
    }

    void P() {
        arbiter.lock(0);
        cout <<"Process P enters" <<endl;
        cout <<"Process P leaves" <<endl;
        arbiter.unlock();
        wait(5, SC_NS);
        arbiter.lock(0);
        cout <<"Process P enters again" <<endl;
        cout <<"Process P leaves again" <<endl;
        arbiter.unlock();
    }

    void Q() {
        arbiter.lock(1);
        cout <<"Process Q enters" <<endl;
        wait(10, SC_NS);
        cout <<"Process Q leaves" <<endl;
        arbiter.unlock();
    }

    void R() {
        arbiter.lock(2);
        cout <<"Process R enters" <<endl;
        wait(SC_ZERO_TIME);
        cout <<"Process R leaves" <<endl;
        arbiter.unlock();
    }
}
```

```
    }  
};  
  
int sc_main(int argc , char *argv[])  
{  
    top TOP("TOP");  
    sc_start(-1);  
    return 0;  
}
```

Résultats attendus

```
Process P enters  
Process P leaves  
Process Q enters  
Process Q leaves  
Process P enters again  
Process P leaves again  
Process R enters  
Process R leaves
```

Annexe B

Test de performance : l'indexeur

L'exemple, dont le code est donné ci-dessous, est présenté à la section [7.2](#).

B.1 Code source complet

```
#include <systemc.h>
#include <vector>
using namespace std;

#define max 4

const int size = 128;
int table[size];

int hash(int w) {
    return (w*7)%size;
}

class top;
top * TOP;

class element : public sc_module
{
public:
    int tid, m, msg, h;
    sc_event msg_event;

    SC_HAS_PROCESS(element);
    element(sc_module_name name, int n) :
        sc_module(name), tid(n), m(0), msg(0), h(1)
    {
        SC_THREAD(T);
        SC_THREAD(getmsg);
    }
}
```

```

void T() {
    wait(20, SC_NS);
    while (1) {
        msg_event.notify();
        wait(msg_event);
        h = hash(msg);
        while (table[h] != 0) {
            h = (h+1) % size;
        }
        table[h] = msg;
    }
}

void getmsg() {
    while (m<max) {
        wait(msg_event);
        msg = (++m) * 11 + tid;
        msg_event.notify();
    }
}

class top : public sc_module
{
public:

    vector<element *> elements;

    top(sc_module_name name, int num) :
        sc_module(name) {
        char s_name[16];

        for (int i=0; i<num; i++) {
            sprintf(s_name, "element_%d", i+1);
            elements.push_back(new element(s_name, i));
        }
    }
};

void dump_table() {
    cout <<"DUMP ";
    for (int n=0; n<size; n++)
        cout <<table[n] <<" ";
    cout <<endl;
}

```

```
int sc_main(int argc , char *argv[])
{
    int num = 2;

    if (argc > 1)
        num = atoi(argv[1]);

    if (num < 1)
        num = 1;
    if (num > 100)
        num = 100;

    TOP = new top("TOP", num);
    sc_start(-1);
    dump_table();
    return(0);
}
```

B.2 Comparaison avec la version instrumentée

```
8c8
< int table[size];
---
> rvs_probe<int> table[size]; //patch
20c20,22
< int tid, m, msg, h;
---
> int tid, m;
> rvs_probe<int> msg; //patch
> int h;
```


Bibliographie

- [ABG⁺05] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. S. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 2005. 3.3.2
- [AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3) :117–126, 1987. 5.5.2.2
- [Ave05] Avery design systems. Simcluster . parallel simulation, 2005. <http://www.avery-design.com/files/docs/SimCluster.pdf>. 9.3.2
- [B⁺96] Michel Berkelaar et al. Lp_solve, 1996. [http://www.cs.sunysb.edu/~sim\\$algorithm/implement/lpsolve/implement.shtml](http://www.cs.sunysb.edu/~sim$algorithm/implement/lpsolve/implement.shtml). 8.3.2
- [BBBD02] G. Berry, L. Blanc, A. Bouali, and J. Dormoy. Top-level validation of system-on-chip in estereel studio. In *HLDVT '02 : Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop (HLDVT'02)*, page 36, Washington, DC, USA, 2002. IEEE Computer Society. 3.3.1
- [BBM06] Ramzi Ben Salah, Marius Bozga, and Oded Maler. On interleaving in timed automata. In Baier and Hermanns [BH06], pages 465–476. 8.5.2
- [BBP89] Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors. *Temporal Logic in Specification, Altrincham, UK, April 8-10, 1987, Proceedings*, volume 398 of *Lecture Notes in Computer Science*. Springer, 1989. 2.1.3.2
- [BDM⁺98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos : A model-checking tool for real-time systems. In *Proc. 1998 Computer-Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag. 8.5.2
- [BFG⁺00] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF : A validation environment for timed asynchronous systems. In *Computer Aided Verification*, pages 543–547, 2000. 8.5.2
- [BGB02] Mike G. Bartley, Darren Galpin, and Tim Blackmore. A comparison of three verification techniques : directed testing, pseudo-random testing and property checking. In *DAC '02 : Proceedings of the 39th conference on Design automation*, pages 819–823, New York, NY, USA, 2002. ACM Press. 3.3
- [BH06] Christel Baier and Holger Hermanns, editors. *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*. Springer, 2006. B.2
- [BJLY98] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *International Conference on Concurrency Theory*, pages 485–500, 1998. 8.5.2

- [BL03] Benedikt Bollig and Martin Leucker. Deciding ltl over mazurkiewicz traces. *Data Knowl. Eng.*, 44(2) :219–238, 2003. 10.2.1.3
- [BNG⁺06] Rabie Ben Atitallah, Smaïl Niar, Alain Greiner, Samy Meftali, and Jean-Luc Dekeyser. Estimating energy consumption for an mp soc architectural exploration. In Werner Grass, Bernhard Sick, and Klaus Waldschmidt, editors, *ARCS*, volume 3894 of *Lecture Notes in Computer Science*, pages 298–310. Springer, 2006. 2.2.3
- [Bou02] F. Boussinot. Java fair threads. Technical report, Inria, 2002. <http://www-sop.inria.fr/mimosa/rp/FairThreads/html/FairThreads.htm>. 4.4.2
- [BR00] Thomas Ball and Sriram K. Rajamani. Boolean programs : A model and process for software analysis. Technical report, Microsoft Research, February 2000. 3.1.1
- [Bro02] Chris Browy. Comparing testwizrd and specman for transaction-level verification. white paper, Avery Design Systems, February 2002. était disponible à l'adresse <http://www.avery-design.com/twvp.html>. 3.2.2
- [Bry77] R. E. Bryant. Simulation of packet communication architecture computer systems. Master's thesis, MIT Laboratory for Computer Science, Cambridge, MA, USA, November 1977. Technical Report TR-188. 9.3.2.1
- [BWH⁺03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis : An integrated electronic system design environment. *Computer*, 36(4) :45–52, 2003. 2.3.1.2
- [Cad99] Cadence Design Systems. NC-SystemC, 1999. http://www.cadence.com/products/functional_ver/nc-systemc/. 2.3.1.1
- [CC04] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2004. 8.2.4.2
- [CC05] Robert Clariso and Jordi Cortadella. Verification of concurrent systems with parametric delays using octahedra. In *ACSD '05 : Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, pages 122–131, Washington, DC, USA, 2005. IEEE Computer Society. 8.5.2
- [CCG⁺02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2 : An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer. 2.1.3.2
- [CCZ06] Bastien Chopard, P. Combes, and J. Zory. A conservative approach to systemc parallelization. In *International Conference on Computational Science (4)*, pages 653–660, 2006. 9.3.2
- [CG03] Lukai Cai and Daniel Gajski. Transaction level modeling : an overview. In *CODES+ISSS '03 : Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM Press. 2.1.2.2
- [CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. 3.1.1

- [CL97] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems*, Newport Beach, California, 1997. 8.5.1
- [CLI⁺03] Franco Carbognani, Christopher K. Lennard, C. Norris Ip, Allan Cochrane, and Paul Bates. Qualifying precision of abstract systemc models using the systemc verification standard. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 20088, Washington, DC, USA, 2003. IEEE Computer Society. 3.2.2
- [clp04] COIN-OR Linear Program Solver (CLP), 2004. <http://www.coin-or.org/Clp/index.html>. 8.3.2
- [CM79] K. Mani Chandy and Jayadev Misra. Distributed simulation : A case study in design and verification of distributed programs. *IEEE Trans. Software Eng.*, 5(5) :440–452, 1979. 9.3.2.1
- [CMMC07] Jérôme Cornet, Florence Maraninchi, and Laurent Maillet-Contoz. Séparation des aspects fonctionnels/non-fonctionnels dans les modèles transactionnels des systèmes sur puce, January 2007. FETCH (poster). 2.2.3
- [Con99a] World Wide Web Consortium. XML Path Language (XPath), 1999. <http://www.w3.org/TR/xpath>. 6.6.1.2
- [Con99b] World Wide Web Consortium. XSL Transformations (XSLT), 1999. <http://www.w3.org/TR/xslt>. 6.6.1.2
- [Con03] World Wide Web Consortium. Scalable Vector Graphics (SVG), 2003. <http://www.w3.org/Graphics/SVG/>. 6.6.1.2
- [Cox05] David Richard Cox. Ritsim : Distributed systemc simulation. Master's thesis, Rochester Institute of Technology, Rochester, New York, august 2005. 9.3.2
- [DG03] Rolf Drechsler and Daniel Große. Formal verification of ltl formulas for systemc designs, 2003. <http://www.informatik.uni-bremen.de/grp/ag-ram/doc/konf/iscas03%verification.systemc.pdf>. 3.1.1
- [DGG⁺05] Anat Dahan, Daniel Geist, Leonid Gluhovsky, Dmitry Pidan, Gil Shapir, Yaron Wolfsthal, Lyes Benalycherif, Romain Kamdem, and Younes Lahbib. Combining system level modeling with assertion based verification. In *ISQED'05 : Proceedings of the 6th International Symposium on Quality of Electronic Design*, pages 310–315, Washington, DC, USA, 2005. IEEE Computer Society. 7.4
- [DGG06] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual, version 2.0*. SpecC Technology Open Consortium, 2006. http://www.ics.uci.edu/~specc/reference/SpecC_LRM_20.pdf. 2.3.1.2
- [DP05] Andrei Danes and Eric Paire. Systemc tlm acceleration, 2005. Company internal - unpublished. 9.3.2
- [EJN⁺02] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin. X-gen : a random test-case generator for systems and socs. In *HLDVT '02 : Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop (HLDVT'02)*, page 145, Washington, DC, USA, 2002. IEEE Computer Society. 3.2.2

- [FFP01] Alessandro Fin, Franco Fummi, and Graziano Pravadelli. Amleto : A multi-language environment for functional test generation. In *ITC '01 : Proceedings of the 2001 IEEE International Test Conference*, page 821, Washington, DC, USA, 2001. IEEE Computer Society. 3.2.2
- [FFS01] Alessandro Fin, Franco Fummi, and Denis Signoreto. The use of systemc for design verification and integration test of ip-cores. In *ASIC/SOC Conference, 2001. Proceedings. 14th Annual IEEE International*, pages 76–80. IEEE Computer Society, 2001. 3.2.2
- [FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05 : Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110–121, New York, NY, USA, 2005. ACM Press. 1.6.1, 3.3.2, 5.1.2, 5.4.2, 7.2.1, 7.2.1, 10.1.2
- [For04] Forte Design Systems. Cynthesizer, 2004. <http://www.forteds.com/products/index.asp>. 2.1.2.4
- [FZ03] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *DAC '03 : Proceedings of the 40th conference on Design automation*, pages 286–291, New York, NY, USA, 2003. ACM Press. 3.3.1
- [GFL⁺96] Daniel Geist, Monica Farkas, Avner Landver, Yossi Lichtenstein, Shmuel Ur, and Yaron Wolfsthal. Coverage-directed test generation using symbolic techniques. In Mandayam K. Srivas and Albert John Camilleri, editors, *FMCAD*, volume 1166 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 1996. 3.3.1
- [GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, 1975. 3.3.2
- [Ghe05] Frank Ghenassia, editor. *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, June 2005. ISBN 0-387-26232-6. 2.2.1
- [GHO⁺98] Raanan Grinwald, Eran Harel, Michael Orgad, Shmuel Ur, and Avi Ziv. User defined coverage - a tool supported methodology for design verification. In *DAC '98 : Proceedings of the 35th annual conference on Design automation*, pages 158–163, New York, NY, USA, 1998. ACM Press. 3.2.1
- [GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart : directed automated random testing. In *PLDI '05 : Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press. 10.2.3
- [glp00] GNU Linear Programming Kit (GLPK), 2000. <http://directory.fsf.org/glpk.html>. 8.3.2
- [GNJ⁺96] Gopi Ganapathy, Ram Narayan, Glenn Jordan, Denzil Fernandez, Ming Wang, and Jim Nishimura. Hardware emulation for functional verification of k5. In *DAC '96 : Proceedings of the 33rd annual conference on Design automation*, pages 315–318, New York, NY, USA, 1996. ACM Press. 2.1.1
- [God96] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems : an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996. 7.1.2

- [God05] Patrice Godefroid. Software model checking : The verisoft approach. *Formal Methods in System Design*, 26(2) :77–101, 2005. [10.2.1.3](#)
- [GSW00] Yuri Gurevich, Wolfram Schulte, and Charles Wallace. Investigating java concurrency using abstract state machines. In *ASM '00 : Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, pages 151–176, London, UK, 2000. Springer-Verlag. [3.1.2](#)
- [HB02] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*, Grenoble, October 2002. LNCS 2491, Springer Verlag. [3.1.2](#)
- [HG96] Ulrich Heinkel and Wolfram H. Glauert. An approach for a dynamic generation/validation system for the functional simulation considering timing constraints, 1996. [8.5.1](#)
- [HGR⁺01] Dirk Hoffmann, Joachim Gerlach, Juergen Ruf, Thomas Kropf, Wolfgang Mueller, and Wolfgang Rosenstiehl. The simulation semantics of systemC, December 21 2001. [3.1.2](#)
- [HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992. [2.1.3.2](#), [3.1.2](#)
- [HMMC06] Claude Helmstetter, Florence Maraninchi, and Laurent Maillet-Contoz. Test coverage for loose timing annotations. In *11th International Workshop on Formal Methods for Industrial Critical Systems*. Springer-Verlag, August 2006. [1.6.1](#), [10.1.3](#)
- [HMMCM06] Claude Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz, and Matthieu Moy. Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. *FMCAD, 0* :171–178, 2006. [1.6.1](#), [10.1.2](#)
- [HPVB00] Klaus Havelund, Seungjoon Park, Willem Visser, and Guillaume Brat. Java pathfinder - second generation of a java model checker. In *In Proc. of Post-CAV Workshop on Advances in Verification*, July 2000. [3.1.1](#), [3.3.2](#)
- [HSH⁺00] Pei-Hsin Ho, Thomas R. Shiple, Kevin Harer, James H. Kukula, Robert F. Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart simulation using collaborative formal and simulation engines. In Ellen Sentovich, editor, *ICCAD*, pages 120–126. IEEE, 2000. [3.3.1](#)
- [HTCT03] Y. Hsu, B. Tabbara, Y. Chen, and F. Tsai. Advanced techniques for RTL debugging, 2003. [2.1.1](#)
- [HZB⁺03] Renate Henftling, Andreas Zinn, Matthias Bauer, Martin Zambaldi, and Wolfgang Ecker. Re-use-centric architecture for a fully accelerated testbench environment. In *DAC '03 : Proceedings of the 40th conference on Design automation*, pages 372–375, New York, NY, USA, 2003. ACM Press. [2.1.1](#)
- [Ip00] C. Norris Ip. Simulation coverage enhancement using test stimulus transformation. In *ICCAD '00 : Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 127–134, Piscataway, NJ, USA, 2000. IEEE Press. [3.3.1](#)
- [Jea03] B. Jeannot. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1) :5–37, July 2003. [3.1.2](#)

- [KGS06] Vineet Kahlon, Aarti Gupta, and Nishant Sinha. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 286–299. Springer, 2006. [10.2.2](#)
- [Kou94] Eleftherios Koutsoufios. Editing graphs with *dotty*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, July 1994. This report, and the program, is included in the `graphviz` package, available for non-commercial use at <http://www.research.att.com/sw/tools/graphviz/>. [6.6.1.1](#)
- [KOW⁺01] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, Marc Edwards, and Yaron Kashi. A framework for object oriented hardware specification, verification, and synthesis. In *DAC '01 : Proceedings of the 38th conference on Design automation*, pages 413–418, New York, NY, USA, 2001. ACM Press. [3.2.2](#), [3.2.2](#)
- [Kup06] Orna Kupferman. Sanity checks in formal verification. In Baier and Hermanns [BH06], pages 37–51. [3](#)
- [LAK98] B. Pradin-Chezalviel L. A. Kunzle, R. Valette. Temporal reasoning in fuzzy time petri nets. Technical Report 98073, LAAS Toulouse, 1998. [8.5](#)
- [LC06] Yu Lei and Richard H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6) :382–403, 2006. [5.1.2](#)
- [LML06] Xiaojun Liu, Eleftherios Matsikoudis, and Edward A. Lee. Modeling timed concurrent systems. In Baier and Hermanns [BH06], pages 1–15. [8.2.2.1](#)
- [LMTY02] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and verifying systems with `tla`, 2002. [2.1.3.2](#)
- [LNZ05] D. Lugiez, P. Niebert, and S. Zennou. A partial order semantics approach to the clock explosion problem of timed automata. *Theor. Comput. Sci.*, 345(1) :27–59, 2005. [8.5.2](#)
- [Man06] Louis Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6, 2006. [4.4.2](#)
- [Maz87] A Mazurkiewicz. Trace theory. In *Advances in Petri nets 1986, part II on Petri nets : applications and relationships to other models of concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc. [5.3.2.2](#)
- [McM92a] K. L. McMillan. The SMV system, November 06 1992. [3.1.2](#)
- [McM92b] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV '92 : Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 164–177, London, UK, 1992. Springer-Verlag. [10.2.1.2](#)
- [Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3) :267–310, July 1983. [3.1.2](#)
- [Min01] Antoine Miné. The octagon abstract domain. In *WCRE*, page 310, 2001. [8.2.4.2](#)
- [MMMC05a] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy : A toolbox for the analysis of systems-on-a-chip at the transactional level. In *International Conference on Application of Concurrency to System Design*, June 2005. [3.1.2](#)
- [MMMC05b] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa : An extraction tool for systemc descriptions of systems-on-a-chip. In *EMSOFT*, September 2005. [6.3.3.1](#)

- [MMMC06] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy : an open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 2006. special issue on SystemC-based systems. 3.1.2, 9.5
- [Moy05] Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005. 3.1.2
- [NS01] Brian Nielsen and Arne Skou. Test generation for time critical systems : Tool and case study. *ecrts*, 00 :0155, 2001. 8.5.1
- [Ope03] Open SystemC Initiative. *SystemC v2.0.1 Language Reference Manual*, 2003. <http://www.systemc.org/>. 4.1.1
- [Ope05] Open SystemC Initiative. *SystemC v2.1 Language Reference Manual (IEEE Std 1666-2005)*, 2005. <http://www.systemc.org/>. 2.3.1.1, 9.1
- [Pag96] Florence Pagani. Partial orders and verification of real-time systems. In *FTRTFT '96 : Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 327–346, London, UK, 1996. Springer-Verlag. 8.5.2
- [Pas02] Sudeep Pasricha. Transaction level modeling of SoC in systemc 2.0. Technical report, STMicroelectronics Ltd, 2002. 2.1.2.2
- [Pe193] Doron Peled. All from one, one for all : on model checking using representatives. In *CAV '93 : Proceedings of the 5th International Conference on Computer Aided Verification*, pages 409–423, London, UK, 1993. Springer-Verlag. 5.1.2
- [Phi03] Ammar Aljer Philippe. BHDL : Circuit design in b, 2003. 3.1
- [PHR04] G. Pace, N. Halbwachs, and P. Raymond. Counter-example generation in symbolic abstract model-checking. *Software Tools for Technology Transfer*, 5(2–3), March 2004. 3.2.2
- [Poo03] Martin Pool. distcc, a fast free distributed compiler. White paper, December 2003. Presented at : linux.conf.au, Adelaide, 2004 ; <http://distcc.samba.org/doc/lca2004/distcc-lca-2004.html>. 10.2.1.3
- [RJR06] Pascal Raymond, Erwan Jahier, and Yvan Roux. Describing and executing random reactive systems. In *SEFM '06 : Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 216–225, Washington, DC, USA, 2006. IEEE Computer Society. 3.2.2, 3.2.2
- [RNHW98] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *RTSS '98 : Proceedings of the IEEE Real-Time Systems Symposium*, page 200, Washington, DC, USA, 1998. IEEE Computer Society. 3.2.2
- [RR02] M. Renaudin and J.-B. Rigaud. Etude de l'art sur la conception des circuits asynchrones, perspectives pour l'intégration des systèmes complexes [TIMA-RR-02/12-02-FR]. Technical report, TIMA, 2002. Document réalisé avec le support de STMicroelectronics, Crolles, janvier 2000. 2.2.2
- [RS03] John Rose and Stuart Swan. SCV randomization, 2003. www.testbuilder.net/reports/scv_randomization.pdf. 3.2.2
- [SA06] Koushik Sen and Gul Agha. jcute : Automated testing of multithreaded programs using race-detection and flipping. Technical Report UIUCDCS-R-2006-2676, University of Illinois at Urbana Champaign, 2006. 5.1.2, 10.2.3

- [Sal03] Ashraf Salem. Formal semantics of synchronous systemc. In *DATE*, pages 10376–10381, 2003. 3.1.2
- [SBN⁺97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser : a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4) :391–411, 1997. 3.3.2
- [SCCS05] Natasha Sharygina, Sagar Chaki, Edmund M. Clarke, and Nishant Sinha. Dynamic component substitutability analysis. In *FM*, pages 512–528, 2005. 3.1.2
- [Sch03a] Klaus-Dieter Schubert. Improvements in functional simulation addressing challenges in large, distributed industry projects. In *DAC '03 : Proceedings of the 40th conference on Design automation*, pages 11–14, New York, NY, USA, 2003. ACM Press. 2.1.1
- [Sch03b] Tom Schubert. High level formal verification of next-generation microprocessors. In *DAC '03 : Proceedings of the 40th conference on Design automation*, pages 1–6, New York, NY, USA, 2003. ACM Press. 2.1.3.2
- [SCI05] STMicroelectronics, CNRS, and INPG. Pinapa, 2005. <http://greensocs.sourceforge.net/pinapa/>. 6.3.3.1
- [SD02] Kanna Shimizu and David L. Dill. Deriving a simulation input generator and a coverage metric from a formal specification. In *Proceedings of the Design Automation Conference*, June 2002. 3.2.2, 3.2.2
- [SPI03] The SPIRIT Consortium. Spirit, 2003. <http://www.spiritconsortium.org>. 2.3.1.2
- [STM07] STMicroelectronics. STMicroelectronics announces world's first dual HDTV decoder chip to be manufactured in 65nm technology, January 2007. Communiqué de presse ; <http://www.st.com/stonline/stappl/press/news/year2007/p2112.htm>. 7.3.5.2
- [tac05] TAC : Transaction accurate communication/channel, 2005. <http://http://www.greensocs.com/TACPackage>. 3.2.2, 4.3.2
- [Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3 :121–189, 1995. 10.2.1.2
- [tlm06] OSCI SystemC TLM 2.0, draft 1 for public review, 2006. http://www.systemc.org/web/sitedocs/TLM_2_0.html. 2.2.1
- [TYB03] Serdar Tasiran, Yuan Yu, and Brannon Batson. Using a formal specification and a model checker to monitor and direct simulation. In *DAC '03 : Proceedings of the 40th conference on Design automation*, pages 356–361, New York, NY, USA, 2003. ACM Press. 3.3.1
- [Upp06] Uppsala and Aalborg Universities. Uppaal, 1994-2006. <http://www.uppaal.com/>. 8.5.2
- [Vil05] Robert Clarisó Viladrosa. *Abstract Interpretation Techniques for the Verification of Timed Systems*. PhD thesis, Universitat Politècnica de Catalunya, June 2005. 8.5.2
- [VPG06] Emmanuel Viaud, François Pécheux, and Alain Greiner. An efficient tlm/t modeling and simulation environment based on conservative parallel discrete event principles. In *DATE '06 : Proceedings of the conference on Design, automation and test in Europe*, pages 94–99, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. 2.2.3

- [Wik06] Wikipédia. Circuit logique programmable, 2006. http://fr.wikipedia.org/w/index.php?title=Circuit_logique_programmable&oldid=12045081. 2.1.1
- [YG02] Jin Yang and Amit Goel. Gste through a case study. In *ICCAD '02 : Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 534–541, New York, NY, USA, 2002. ACM Press. 2.1.3.2
- [YKM02] Tomohiro Yoneda, Tomoya Kitai, and Chris J. Myers. Automatic derivation of timing constraints by failure analysis. In *CAV '02 : Proceedings of the 14th International Conference on Computer Aided Verification*, pages 195–208, London, UK, 2002. Springer-Verlag. 8.5.1
- [YR99] Tomohiro Yoneda and Hiroshi Ryu. Timed trace theoretic verification using partial order reduction. In *ASYNC '99 : Proceedings of the 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 108, Washington, DC, USA, 1999. IEEE Computer Society. 8.5.1
- [YS01] J. Yang and C.-J. Seger. Introduction to generalized symbolic trajectory evaluation. In *ICCD '01 : Proceedings of the International Conference on Computer Design : VLSI in Computers & Processors*, page 360, Washington, DC, USA, 2001. IEEE Computer Society. 2.1.3.2
- [Zen04] Sarah Zennou. *Méthodes d'ordre partiel pour la vérification de systèmes concurrents et temps réel*. PhD thesis, Université Aix-Marseille I, France, December 2004. 8.5.2

Index

- PV.wait, [14](#), [124](#)
- SC.METHOD, [44](#)
- SC.THREAD, [44](#)
- sc.module, [29](#)
- δ -cycle, [44](#), [53](#), [127](#), [145](#)

- abstraction conservatrice, [35](#)
- accès mémoire, [90](#), [93](#)
- action, [143](#)
- action de communication, [46](#), [66](#)
- algorithme du Simplex, [132](#)
- algorithme principal, [70](#), [131](#)
- analyse statique, [151](#)
- analyseur, [92](#)
- annotation temporelle, [27](#), [124](#)
- approche structurelle, [146](#)
- arbitrage, [50](#), [52](#)
- arbre des contraintes d'ordonnancement, [74](#), [76](#),
[101](#)
- architecture, [25](#), [29](#)
- aspect, [27](#)
- assistant de preuve, [36](#)
- asynchrone, [11](#), [27](#), [36](#), [44](#)
- attente, [44](#), [46](#)

- backtracking, [158](#)
- bibliographie, [16](#)
- BIST, [24](#)
- byte enable, [26](#)

- canal de communication, [29](#)
- carte des adresses mémoires, [26](#)
- changement de contexte, [44](#)
- cible, [26](#)
- commutativité, [66](#)
- connexion directe, [30](#)
- consommation, [28](#)
- contrôleur des sorties, voir oracle
- Contrainte d'ordonnancement, [62](#)
- contrainte héritée, [72](#)

- contrainte temporelle, [131](#)
- cosimulation, [20](#)
- couverture, [12](#), [38](#)

- data-race, [47](#)
- date symbolique, [130](#)
- deadlock, voir interblocage
- délai borné, [14](#), [124](#)
- dépendance, [60](#), [143](#)
- dépendance à l'ordonnancement, [12](#), [42](#), [52](#)
- dépendance à la temporisation, [124](#)
- désactivation, [78](#)
- design gap, [19](#)
- dépendance à l'ordonnancement, [47](#)

- e (langage), [39](#)
- écoulement du temps, [44](#)
- égalité de transition, [64](#)
- éligible, [44](#)
- émulation matérielle, [21](#)
- énergie, [28](#)
- enregistreur, [87](#)
- équivalence d'exécutions, [42](#)
- équivalence d'ordonnements, [13](#), [59](#)
- erreur de synchronisation, [12](#)
- erreur locale, [79](#)
- esclave, voir cible
- état global, [58](#), [142](#)
- évaluation de performances, [27](#)
- événement persistant, [50](#), [114](#), [156](#)
- événement SystemC, [68](#), [87](#)
- explosion combinatoire, [12](#), [107](#)

- feuille morte, [74](#), [104](#)
- file à priorités, [48](#)
- flot de contrôle, [40](#), [151](#)
- fonction de classement, [77](#)
- FPGA, [21](#)

- gel des processus, [73](#)

- générateur d'entrées, 37, 58, 122
générateur pour TLM, 40
génération d'ordonnancements, 13, 56, 70, 96
génération de jeux de durées, 14, 128
granularité, 27
graphe des dépendances dynamiques, 63, 97, 129
graphe des dépendances statiques, 62, 97
- HPIOM, 36
- impasse, voir feuille morte
implantation, 13, 81, 134
implantation OSCI, 12, 44, 48, 83
indépendance, voir dépendance
indexeur, 107
initiateur, 26
insertion de code, 89
instrumentation, 13, 87, 111, 116, 152
interblocage, 48, 50, 78
interface composant, 33
interface processus, 33
IP, voir propriété intellectuelle
ISS, 20
- jeu complet de contraintes, 77
jeu de données, 57
jeu de durées, 14, 131, 135
- LCMPEG, 110
légende des identifiants, 86
librairie TLM, 30
limitations, 58, 69, 126
liste d'événements, 69
localement asynchrone, 44
logiciel embarqué, 20
logiciel libre, 28, 83
Lurette, 39
LusSy, 10, 36, 41, 137, 158
Lustre, 36
- maître, voir initiateur
masque, 20
Metropolis, 29
migration de processus, 44
mise à jour synchrone, 53, 145
modèle de référence, 11, 24
modèle fonctionnel, 22, 27
modèle temporisé, 23, 27, 123
- modélisation transactionnelle, voir TLM
module, 29
- NC-SystemC, 28
net unfolding, 157
niveau algorithme, 22
niveau cycle accurate, 23
niveau portes logiques, 23
niveaux d'abstraction, 21
non-préemptivité, 44, 141, 147
notification, 46
notification dépendante, 68
notification retardée, 44
- observateur, voir oracle
Open SystemC Initiative, voir OSCI
oracle, 38, 79, 114, 158
ordonnancement, 11, 43, 58, 140
ordonnancement indéterministe, 12, 44
ordonnancement valide, 58
ordonnanceur aléatoire, 12, 83
ordonnanceur instable, 48
ordonnanceur interactif, 83
ordre causal, 69, 93
OSCI, 10
- parallélisation, 15, 141, 158
partitionnement logiciel - matériel, 20
passage à l'échelle, 140
passé d'une transition, 64
permutabilité, 61, 69
permutation, 59
phase d'élaboration, 30, 44
phase d'évaluation, 44
phase de mise à jour, 44
physiquement idéal, 27
Pinapa, 88
polling, 111, 114
preuve, 77
preuve par construction, 27
processus, 11, 44
programmation linéaire, 135
programme linéaire, 130
Programmer View, voir PV
Programmer View + Timing, voir PVT
propriété intellectuelle, 20, 39
propriété principale, 75
protocole, 30

- prototype, 81
PV, 27
PVT, 27
- raffinement, 21
réduction d'ordre partiel, 13, 42, 56, 158
représentativité, 53
résultats, 112, 116, 136
router, 33
RTL, 9, 20, 23
runtime verification, 42
- schéma de temporisation, voir jeu de durées
SCV, 40
sémantique formelle, 37
simulateur parallèle, 150
simulation, 20
simulation distribuée, 146
SMV, 36
SoC, voir système sur puce
SpecC, 29
spécification SystemC, 12, 44, 141
SPG, 10
SST, voir système sous test
SSTD, 58
standard IEEE, 28
STMicroelectronics, 10
synchrone, 36
synthèse, 37
System Platform Group, voir SPG
SystemC, 10, 28
SystemC Verification, voir SCV
système sous test, 37, 58
système sur puce, 9, 20
- TAC, 33
TAC_router, voir router
temps imprécis, 14, 124
temps local, 28
temps simulé, 44
test boîte grise, 91
test dirigé, 41
test logiciel, voir validation par simulations
test matériel, 24
tissage dynamique, 27
TLM, 10, 21, 22, 25
tlm_synchro, 33
trace d'exécution, 85
trace détaillée, 99
transaction, 26
transfert de registre, voir RTL
transition, 15, 59, 64, 143
trou de couverture, 12, 39
- validation par simulations, 11, 37
variable partagée, 46, 67
vérification à la volée, 42
vérification compositionnelle, 37
vérification formelle, 12, 24, 35, 137
vérification semi-formelle, 40
Verimag, 10
verrou, 51
vitesse de simulation, 21, 139
- Xerces, 92
XSLT, 98
- yield, 46