

Université de Montréal

**Application du concept des transactions pour la modélisation et la
simulation multicœur des systèmes sur puce**

par
Amine Anane

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Janvier, 2012

© Amine Anane, 2012.

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée:

**Application du concept des transactions pour la modélisation et la
simulation multicœur des systèmes sur puce**

présentée par:

Amine Anane

a été évaluée par un jury composé des personnes suivantes:

Stefan Monnier,	président-rapporteur
El Mostapha Aboulhamid,	directeur de recherche
Yvon Savaria,	codirecteur
Abdelhakim Hafid,	membre du jury
Samy Meftali,	examineur externe
Clément Arsenault,	représentant du doyen de la FAS

Thèse acceptée le:



Résumé

Avec la complexité croissante des systèmes sur puce, de nouveaux défis ne cessent d'émerger dans la conception de ces systèmes en matière de vérification formelle et de synthèse de haut niveau. Plusieurs travaux autour de SystemC, considéré comme la norme pour la conception au niveau système, sont en cours afin de relever ces nouveaux défis. Cependant, à cause du modèle de concurrence complexe de SystemC, relever ces défis reste toujours une tâche difficile. Ainsi, nous pensons qu'il est primordial de partir sur de meilleures bases en utilisant un modèle de concurrence plus efficace. Par conséquent, dans cette thèse, nous étudions une méthodologie de conception qui offre une meilleure abstraction pour modéliser des composants parallèles en se basant sur le concept de transaction. Nous montrons comment, grâce au raisonnement simple que procure le concept de transaction, il devient plus facile d'appliquer la vérification formelle, le raffinement incrémental et la synthèse de haut niveau. Dans le but d'évaluer l'efficacité de cette méthodologie, nous avons fixé l'objectif d'optimiser la vitesse de simulation d'un modèle transactionnel en profitant d'une machine multicœur. Nous présentons ainsi l'environnement de modélisation et de simulation parallèle que nous avons développé. Nous étudions différentes stratégies d'ordonnancement en matière de parallélisme et de surcoût de synchronisation. Une expérimentation faite sur un modèle du transmetteur Wi-Fi 802.11a a permis d'atteindre une accélération d'environ 1.8 en utilisant deux *threads*. Avec 8 *threads*, bien que la charge de travail des différentes transactions n'était pas importante, nous avons pu atteindre une accélération d'environ 4.6, ce qui est un résultat très prometteur.

Mots-clés : Modélisation, Simulation parallèle, Transactions, Multi-cœurs



Abstract

With the increasing complexity of SoCs, new challenges continue to emerge in the design of these systems in terms of formal verification and high-level synthesis. Several research efforts around SystemC, considered the de facto standard for system-level design, are underway to meet these new challenges. However, because of the complex concurrency model of SystemC, these challenges remain difficult tasks. Thus, we believe it is important to continue on a better footing by using a more effective concurrency model. Therefore, in this thesis, we study a design methodology that provides a better abstraction for modeling parallel components based on the concept of transaction. We show how, through simple reasoning about transactions, it becomes easier to apply formal verification, incremental refinement and high-level synthesis. In order to evaluate the effectiveness of this methodology, we set the goal to optimize the simulation speed of a transactional model by taking advantage of a multicore machine. We present a modeling and parallel simulation environment that we developed. We study different scheduling strategies in terms of parallelism and synchronization overhead. An experiment made on a Wi-Fi 802.11a transmitter model achieved a speed up of about 1.8 using two threads. With 8 threads, although the workload of individual transactions was not significant, we could reach a speed up equal to 4.6 which is a very promising result.

Keywords: SOC Design, Parallel Simulation, Transactions, Multi-core



Table des matières

Résumé	iii
Abstract	iv
Table des matières	v
Liste des figures	viii
Liste des codes source	xi
Liste des Sigles	xii
Remerciements	xiii
Chapitre 1 : Introduction	1
1.1 Motivations	1
1.2 Méthodologie	5
1.3 Objectifs	8
1.4 Contributions	9
1.5 Plan	12
Chapitre 2 : Conception des SOC	14
2.1 Flot de conception classique et ses limitations	15
2.2 Modélisation au niveau transaction (TLM)	18
2.3 SystemC et ses limitations	24
2.3.1 Vue d'ensemble	25
2.3.2 Limitations de SystemC	28

Chapitre 3 : Les transactions	34
3.1 Mémoire transactionnelle (TM)	34
3.1.1 Avantages de TM	34
3.1.2 Implémentation de TM	38
3.2 Synthèse de matériel	41
3.2.1 Implémentation de référence	42
3.2.2 Optimisations	46
3.2.3 Bluespec	49
3.3 Raisonnement formel	51
3.3.1 Aperçu de TLA	51
3.3.2 Raffinement	55
Chapitre 4 : Validité d’une exécution concurrente	57
4.1 Environnement d’exécution	57
4.2 Sérialisabilité	61
4.3 Syntaxe d’un ordonnancement	64
4.4 Sémantique d’un ordonnancement	66
4.5 Sérialisabilité “Final State”	70
4.6 Sérialisabilité “View”	72
4.7 Sérialisabilité conflictuelle	75
4.8 Stratégies d’ordonnancement	79
Chapitre 5 : Environnement de modélisation et simulation	84
5.1 Modèle transactionnel	86
5.2 Analyse statique	89
5.2.1 Élaboration	89
5.2.2 Analyse des transactions	91
5.2.3 Construction du graphe de conflit	93
5.3 Ordonnancement	95
5.3.1 Ordonnancement dynamique	99
5.3.2 Limiter le nombre de transactions	101

5.3.3	Ordonnancement statique	103
5.4	Mise en œuvre en utilisant .NET	105
Chapitre 6 : Expérimentation		112
6.1	Modèle du transmetteur 802.11a	112
6.1.1	Contrôleur	113
6.1.2	Embrouilleur	113
6.1.3	Encodeur convolutionnel	113
6.1.4	Entrelaceur	114
6.1.5	Mappeur	115
6.1.6	IFFT	116
6.1.7	Extenseur de cycle	116
6.2	Résultats expérimentaux	116
6.2.1	Cadre expérimental	116
6.2.2	Résultats expérimentaux	118
6.2.3	Implémentations et tests	126
6.2.4	Optimisation de l'ordonnancement	135
Chapitre 7 : Conclusion		148
7.1	Bilan	148
7.2	Perspectives	152
Bibliographie		156
Annexe I : Raffinement d'une FIFO		xiv
I.1	Interface	xv
I.2	Raffinement des données	xvii
I.3	Raffinement de l'atomicité	xviii
I.4	Implémentation finale	xx

Liste des figures

1.1	Environnement de Conception	10
2.1	Flot de conception classique	15
2.2	Modèles TLM	21
2.3	Architecture du langage SystemC	26
2.4	Modèle d'exécution de SystemC	27
3.1	Difficulté d'étendre le logiciel avec des verrous[24]	37
3.2	Circuit RTL d'un registre correspond à une variable accédée par deux transactions	43
3.3	Circuit RTL généré à partir d'un modèle transactionnel	44
3.4	Circuit généré à partir du modèle transactionnel $\text{gcd}(p, q)$	45
3.5	Analyse des transactions. (a) graphe $<_{SC}$ orienté, (b) graphe $<_{SC}$ acyclique, (c) groupe d'arbitrages	49
4.1	Sémantique d'Exécution d'un Modèle Transactionnel	58
4.2	Environnement d'exécution d'un modèle de transactions.	59
4.4	Exemple de Multigraphe de conflit.	80
4.5	Correspondance entre multigraphe de conflit et graphe de sériali- sabilité.	81
4.6	Exécution Concurrente	82
5.1	Environnement de Modélisation et Simulation	85
5.3	Phase de l'Analyse Statique	90
5.4	Graphe de Relation de l'Exemple 5.2	92
5.5	Graphe de Conflit de l'Exemple 5.2	94

5.6	Ordonnancement des transactions	96
5.7	Exemple de coloration de graphe	96
5.8	Nouvelle coloration du graphe de la Figure 5.7	97
5.9	Un exemple de Thread Pool.	100
5.10	Thread Pool avec nombre de tâches ne dépassant pas le nombre de <i>threads</i>	104
6.1	Modèle du transmetteur 802.11a.	112
6.2	Fonction d'embrouillage	114
6.3	Fonction d'encodage	114
6.4	Coloration du Graphe de Conflit	119
6.5	Exécution Parallèle	119
6.6	Temps d'exécution suite à l'ordonnancement de la Figure 6.5	120
6.7	Comparaison des Temps d'exécution obtenus par rapport aux meilleurs temps que nous pouvons obtenir.	124
6.8	Un étage parmi les trois étages de la nouvelle IFFT Pipelinée. . . .	125
6.9	Temps d'exécution en utilisant l'architecture pipelinée de la Figure 6.8	126
6.10	Pertes dues aux surcoûts.	129
6.11	Surcoût d'ordonnancement.	130
6.12	Surcoût d'interférence.	132
6.13	Attente du Garbage Collector pour finir le ramassage.	133
6.14	Comparaison des surcoûts d'interférence selon le type du Garbage Collector.	134
6.15	Ordonnancement des transactions en présence de verrous.	137
6.16	Associer un seul verrou pour les trois transactions du graphe de verrous au lieu d'un verrou pour chaque paire.	137
6.17	Multigraphe de Conflit	141
6.18	Graphe de Concurrence et Attribution des Verrous.	142
6.19	Relations de conflits lors de l'accès à la FIFO	143
6.20	Exemple de FIFO Concurrente	144

6.21	Nouveau multigraphe de conflit ayant moins de cycles et de verrous	145
6.22	Temps d'exécution par abstraction de la FIFO	146
7.1	Environnement de modélisation et simulation	150
7.2	Flot de conception utilisant la méthodologie basée sur les transaction.	152
I.1	Interaction de la FIFO avec son environnement	xvi



Liste des codes source

5.1	Code de l'ordonnanceur	105
5.2	Encapsulation de la méthode avec l'attribut <code>Transaction</code> dans une méthode <code>transaction</code>	106
5.3	Création d'un delegate <code>genericTransaction</code>	108
6.1	Déterminer une estimation du temps d'exécution	123



Liste des Sigles

CSR	Classe des ordonnancements “sérialisables conflictuels”
FSR	Classe des ordonnancements sérialisables “Final State”
IP	Intellectual Property
ITRS	International Technology Roadmap for Semiconductors
MPSOC	MultiProcessor System-on-Chip
OSCI	Open SystemC Initiative
RTL	Register Transfer Level
STM	Software Transactional Memory
SOC	System On a Chip
TLM	Transaction Level Modeling
TM	Transactional Memory
VSR	Classe des ordonnancements sérialisables “View”



Remerciements

Je tiens à remercier en tout premier lieu mon directeur de recherche, le professeur El Mostapha Aboulhamid, de m'avoir soutenu, conseillé et dirigé tout au long de cette thèse.

Je voudrais ensuite remercier toutes les personnes qui ont contribué, de près ou de loin, à ce travail, spécialement Julie Vachon et mon codirecteur le professeur Yvon Savaria.

Je tiens à saluer ma mère, mon père et ma sœur pour m'avoir encouragé à aller au bout de cette thèse.

Finalement, je voudrais remercier ma femme, pour son soutien et sa patience et d'avoir supporté mes sautes d'humeur durant toutes ces années. Sans oublier bien sûr mon cher fils qui était toujours compréhensif quand des fois je le prive de sa tournée au parc pour finir mon travail.

Introduction

En raison de l'avancée fulgurante dans la technologie de fabrication des circuits intégrés qui permet d'intégrer de plus en plus de transistors dans un même circuit, plusieurs composants électroniques, connus sous le nom d'IP, peuvent maintenant être programmés et regroupés dans une même puce afin de former un système électronique complet, appelé système sur puce (SOC).

Cette complexité croissante des systèmes sur puce rend leur conception de plus en plus difficile. Cette difficulté se manifeste par ce qui est nommé, "**écart de productivité**" («*design productivity gap*»). Cet écart découle du fait que le nombre de transistors disponibles augmente plus rapidement que notre capacité à les concevoir efficacement.

Nous allons commencer par montrer les défis que la conception des systèmes sur puce devrait relever afin de minimiser cet écart de productivité. Ensuite nous présentons l'approche de conception que nous avons proposée dans le but de pouvoir relever ces défis. En troisième partie, nous précisons les objectifs que nous envisageons d'atteindre au cours de ce projet, puis nous décrivons les contributions apportées. Finalement, nous donnons le plan de la présente thèse.

1.1 Motivations

Au fil de l'évolution des systèmes microélectroniques industriels, les concepteurs ont été toujours forcés d'améliorer leur productivité afin d'exploiter pleinement la capacité croissante des circuits intégrés et ainsi réduire l'écart de productivité. L'approche utilisée visait toujours l'augmentation du niveau d'abstraction du modèle utilisé pour la description du système à implémenter. Ainsi, la représentation d'un système matériel est passée d'une représentation au niveau transistors

à une représentation composée de portes logiques pour enfin arriver à une représentation basée sur les transferts de registres (RTL). Aujourd'hui, l'utilisation des modèles RTL comme base de départ dans le flot de conception des systèmes sur puce n'offre plus un niveau d'abstraction suffisant pour arriver à réduire l'écart entre la productivité de conception et la capacité d'intégration des puces actuelles. Par conséquent, l'élaboration d'une nouvelle méthodologie de conception initiée à un niveau d'abstraction supérieure à RTL, connue sous le nom de conception au niveau système, est devenue primordiale si l'on souhaite réduire cet écart.

Depuis plus d'une décennie, plusieurs compagnies industrielles ont joint leur effort en formant l'organisme OSCI (Open SystemC Initiative) afin de promouvoir et adopter une nouvelle méthodologie de conception au niveau système connue sous le nom de modélisation au niveau transaction[14] (TLM : Transaction-Level Modeling). Cet effort a abouti à la création du langage de spécification SystemC, qui est devenu une norme IEEE en 2005[34], et à la spécification d'une norme TLM[35] qui régit la communication entre les modèles TLM. Les avantages clefs derrière la méthodologie TLM et le langage SystemC sont les suivants :

- Promouvoir une méthodologie de conception de système sur puce, qui soit adoptée par l'industrie des systèmes microélectroniques afin de rendre possibles l'interopérabilité des modèles et la réutilisation de propriétés intellectuelles (IP).
- Permettre de spécifier une plateforme virtuelle pour commencer à développer et vérifier le logiciel au tout début de cycle de développement.
- Permettre la séparation entre la communication et la fonctionnalité afin de promulguer la réutilisabilité et l'exploration architecturale.

Malgré que la conception au niveau système et l'adoption de la méthodologie SystemC-TLM ont pu faciliter la conception des systèmes sur puce, il existe encore des défis à relever à court et moyen termes puisque la complexité de ces systèmes ne cesse d'augmenter. Voici les différents défis majeurs afin de faire face à cette complexité croissante.

- **Verification Formelle** : L'ITRS («*International Technology Roadmap for Semiconductors*»), un organisme regroupant les principales régions de fabrication des circuits intégrés dans le monde, a été créé afin d'étudier et estimer les besoins technologiques futurs de l'industrie des semiconducteurs sur une période de 15 années. Concernant la conception niveau système, l'ITRS considère que l'écart de productivité est dû en grande partie aux méthodes de vérification actuelles, basées essentiellement sur la simulation, qui nécessitent un temps et un nombre d'ingénieurs considérable afin de pouvoir produire un système de qualité acceptable. Ceci a mené l'ITRS à identifier certains défis à relever concernant les méthodes de vérification. Voici le texte intégral de [36] montrant les besoins à court et à long terme afin de résoudre les problèmes liés aux techniques de vérification des systèmes sur puce :

« Many of the key challenges for verification are relevant to most or all system drivers. In the near term, the primary issues are centered on making formal and semi-formal verification techniques more reliable and controllable. In particular, major advances in the capacity and robustness of formal verification tools are needed, as well as meaningful metrics for the quality of verification. In the longer term, issues focus mainly on raising the level of abstraction and broadening the scope of formal verification. These longer-term issues are actually relevant now, although they have not reached the same level of crisis as other near-term challenges. In general, all of the verification challenges apply to SoC. »

Comme la conception se déplace à un niveau d'abstraction au-dessus de RTL, la vérification devra suivre aussi. En effet, les défis consisteront à :

- adapter et développer des méthodes de vérification formelle à des niveaux d'abstraction supérieurs, pour faire face à la complexité accrue des systèmes rendue possible grâce à la conception au niveau système
- développer des moyens pour vérifier l'équivalence entre les modèles à des niveaux d'abstraction supérieurs et les modèles à des niveaux inférieurs. Ces défis seront beaucoup plus difficiles si les décisions concer-

nant l'augmentation du niveau d'abstraction sont prises sans égard pour la vérification

Par exemple, l'utilisation de langage de description à haut niveau d'abstraction avec des sémantiques mal définies ou inutilement complexes, ou l'adoption d'une méthodologie s'appuyant sur des modèles de simulation à des niveaux d'abstraction supérieurs sans avoir une relation claire avec les niveaux d'abstraction de plus bas niveaux.

Malheureusement SystemC fait partie des langages de description qui ne possèdent pas une sémantique formelle complète et précise afin de pouvoir développer des techniques de vérification formelle efficaces pour celui-ci[60]. Cette difficulté à définir une sémantique formelle pour SystemC provient à la fois de sa nature orientée objet et à la sémantique complexe de son simulateur événementiel[60].

- **Vitesse de simulation :** Comparée à un modèle RTL, la simulation d'un modèle TLM est largement améliorée permettant ainsi l'exploration architecturale et le développement de plateformes virtuels pour la simulation du logiciel. Cependant, avec la complexité croissante des systèmes sur puce et l'augmentation de la quantité de logiciels embarqués, le besoin d'augmenter la vitesse de simulation est de plus en plus ressenti. La disponibilité croissante des machines multicœurs est un atout considérable afin d'améliorer la vitesse de simulation. Toutefois, il faut que la simulation soit conçue pour qu'elle s'exécute en parallèle et ainsi profiter de plusieurs cœurs. Malheureusement le modèle de concurrence basé sur des *threads* synchronisés par des événements rend la parallélisation du simulateur de SystemC une tâche aussi difficile que de lui définir une sémantique formelle.
- **Synthèse de haut niveau (synthèse comportementale) :** Pouvoir synthétiser un système matériel à partir d'un modèle défini à un niveau d'abstraction plus haut que RTL est indispensable pour la conception au niveau système. Malheureusement, malgré que la synthèse comportementale ait été

un sujet de recherche depuis plusieurs décennies, et malgré les progrès récents dus à SystemC et à la modélisation au niveau TLM, des techniques de synthèse comportementale efficaces ne sont pas encore disponibles[36]. En effet, une fois qu'un modèle TLM basé sur SystemC a été spécifié et validé, il doit être raffiné vers un modèle RTL pour la synthèse matérielle. Ce processus est loin d'être automatisé et le raffinement TLM à RTL reste en général une tâche manuelle. En plus, les deux modèles sont parfois préparés par deux ingénieurs différents, qui rend la tâche de raffinement un processus encore plus compliqué[23]. Cette complexité est due principalement à l'écart entre les modèles TLM et RTL où le premier se base sur des appels de fonction alors que l'autre doit gérer et synchroniser la communication entre les différents composants à l'aide de signaux.

1.2 Méthodologie

Afin de faire face à la complexité croissante des systèmes sur puce et relever les différents défis présentés plus haut, nous avons choisi d'utiliser une nouvelle méthodologie qui tient compte des avantages de SystemC tout en évitant ses limitations. Cette méthodologie se base sur le concept des transactions permettant de décrire un système comme étant un ensemble d'actions atomiques. Cette notion fut introduite dans le langage des commandes gardées de Dijkstra[21]. Cependant, avec la complexité croissante des systèmes matériels et logiciels et la disponibilité croissante des machines multicœurs, les recherches actuelles autour des transactions ont pris de l'importance en raison de la nécessité d'avoir une meilleure abstraction afin de décrire des processus parallèles[42]. En conséquence, nous allons étendre ce concept au domaine de la conception des systèmes sur puce.

Dans ce qui suit, nous allons présenter les différents avantages que nous pouvons tirer de cette méthodologie tout en montrant comment elle pourrait éliminer les limitations de SystemC mentionnées précédemment.

- **Raisonnement simple** : Une transaction permet d'abstraire un ensemble d'opérations en une seule action indivisible qui se comporte d'une manière

atomique. Ceci permet au concepteur de raisonner sur la validité de la fonctionnalité d'une transaction comme si elle s'exécute dans un environnement monothread. Ainsi il n'a pas à se préoccuper des interactions complexes avec les autres transactions du programme qui s'exécutent en parallèle.

- **Modélisation à plusieurs niveaux d'abstraction :** Le fait que le modèle de concurrence basé sur les transactions ait un raisonnement simple n'empêche pas qu'il soit assez expressif afin de modéliser des systèmes à plusieurs niveaux d'abstraction. Par exemple, à haut niveau d'abstraction, un processeur peut être spécifié par une simple transaction qui décode et exécute les différentes instructions. En descendant plus bas dans le flot de conception, l'architecture pipelinée du processeur peut être représentée par plusieurs transactions où chaque transaction décrit un étage du pipeline. Par conséquent le fait d'utiliser le même modèle transactionnel tout le long du flot de conception fournit plusieurs avantages tels que :
 - Adopter une méthodologie basée sur le raffinement incrémental où le détail est ajouté progressivement afin de contrôler la complexité du modèle. Ainsi, dans un même modèle transactionnel, nous aurons la possibilité de spécifier différents composants ayant des niveaux d'abstractions différents. Cependant, lors de la simulation, vérification ou synthèse, le raisonnement est toujours le même puisque le même modèle de concurrence est utilisé partout.
 - Le modèle transactionnel s'adapte bien avec la méthodologie basée sur TLM, puisque par nature il est transactionnel. En plus, il donne au terme transaction une sémantique précise comparée à SystemC qui traite une transaction simplement comme un appel de fonction sans tenir compte de l'aspect atomique de la transaction.
 - Contrairement à SystemC où la synthèse matérielle doit passer par un modèle RTL qui est en général spécifié manuellement, la méthodologie basée sur les transactions a pu faire un pas en avant en permettant de

générer automatiquement un modèle RTL à partir d'un modèle transactionnel[32, 31]. Aujourd'hui nous retrouvons Bluespec qui se base des modèles transactionnels pour la synthèse matérielle[12].

- **Vérification Formelle :** Vu la simplicité de la sémantique d'une transaction qui se repose sur sa propriété d'atomicité, plusieurs formalismes ont été proposés afin de pouvoir raisonner formellement sur les transactions. Nous mentionnons à titre d'exemple la logique temporelle des actions (TLA⁺)[40], le langage formel UNITY[50] ou les commandes gardées de Dijkstra[21]. Comme mentionné plus haut, un des défis à relever est de rendre les méthodes de vérification formelle plus fiables et contrôlables. Par conséquent une méthodologie qui se base sur des modèles de calcul ayant une sémantique formelle est une condition nécessaire afin d'atteindre ces objectifs. Grâce à cette sémantique formelle, nous pouvons envisager des techniques de raffinement qui garantissent la génération d'un modèle correct par construction. Ces techniques permettront, soit de raffiner automatiquement un design décrit à un haut niveau d'abstraction vers une implémentation plus concrète, soit de prouver formellement qu'un raffinement fait manuellement préserve les fonctionnalités du modèle initial. Grâce à cette sémantique formelle, il devient possible de développer des algorithmes de transformation de modèles transactionnels ciblant des plateformes de simulation multicœur dans le but d'accélérer le temps de simulation. Ceci constituait l'objectif majeur de ce projet de recherche (voir prochaine section 1.3).
- **Amélioration de la productivité du logiciel embarqué :** La complexité des systèmes augmente considérablement avec l'augmentation de la quantité de logiciels dans les systèmes sur puce et de l'adoption rapide des architectures SOC multicœur (MPSOC). Ainsi une meilleure abstraction pour décrire des systèmes concurrents est nécessaire afin d'améliorer la productivité dans le développement du logiciel embarqué. Comme solution à ce problème, l'ITRS[36] a proposé d'approfondir les recherches sur l'approche de mémoire transactionnelle (TM[42]) afin qu'elle soit adoptée comme méca-

nisme de contrôle de concurrence. Par conséquent l'utilisation d'un modèle transactionnel pour spécifier le logiciel embarqué d'un système sur puce et qui restera le même mécanisme de concurrence dans l'implémentation finale est un atout majeur pour améliorer la productivité des logiciels embarqués.

1.3 Objectifs

Au cours de ce projet de recherche, nous nous sommes fixé l'objectif d'accélérer la vitesse de simulation d'un système transactionnel. Pour aboutir à cet objectif, il faut traiter deux aspects dans le cycle de développement des systèmes sur puce. L'aspect modélisation et l'aspect simulation :

- **Modéliation** : Le premier objectif consiste à développer un environnement de conception basé sur les transactions. Il est important de noter, en comparaison avec SystemC, que la structure du modèle doit être complètement dissociée du simulateur. Ceci est primordial pour que nous puissions adapter le simulateur au modèle afin d'optimiser la vitesse de simulation. Il s'agit d'ailleurs de l'objectif principal de cette thèse décrit plus bas. La première question qui se pose est : quel langage faut-il utiliser afin de décrire des modèles transactionnels ? Nous avons décidé d'utiliser la plateforme de développement «*.Net Framework*» à cause des avantages multiples de cette plateforme tels que la programmation par attribut, l'introspection et la production dynamique de code. Une fois que l'environnement de programmation est choisi, la principale tâche à réaliser est de définir 1) la structure à donner à la description d'un système sur puce, 2) les éléments de cette structure, et 3) la manière de rassembler ces éléments pour former un modèle transactionnel complet.
- **Simulation** : Une des raisons d'élever le niveau d'abstraction à un niveau TLM dans le flot de conception des systèmes sur puce est de permettre le développement du logiciel très tôt dans le cycle de développement. En effet, un modèle TLM va fournir au concepteur de logiciel un prototype virtuel décrivant l'architecture de la plateforme cible. Ce prototype virtuel constitue le

modèle de référence pour développer, simuler et valider le logiciel. Cependant avec l'adoption rapide des architectures SOC multicœur et l'augmentation de la quantité du logiciel embarqué, accélérer la vitesse de simulation devient crucial afin de pouvoir valider le logiciel et avoir une meilleure couverture. Puisque les SOC ont opté pour des architectures multicœurs et vu que le nombre de coeurs dans les stations de travail ne cesse d'augmenter, il est clair que pour minimiser l'écart de productivité il faut suivre la tendance et développer des environnements de conception qui doivent tenir compte des ces nouvelles technologies. En conséquence, l'autre objectif principal de cette thèse est de concevoir des ordonnanceurs de simulation de modèles transactionnels qui prennent avantage d'une machine multicœur afin d'accélérer la vitesse de simulation.

1.4 Contributions

Suite à ce projet, nous pensons avoir contribué au domaine de la conception des systèmes sur puce sur différents points cités ci-dessous.

- L'identification des limitations de SystemC sur différents aspects tels que modélisation, simulation et vérification (voir sous-section 2.3.2) et la proposition d'une nouvelle méthodologie de conception basée sur les transactions[3] dans le but de contourner les obstacles de SystemC et relever les défis identifiés par l'ITRS (voir section 1.1).
- Avoir associé plusieurs recherches dans des domaines différents (chapitre 3) qui de loin semblent toucher des concepts différents alors qu'elles traitent la même notion de transaction. Par exemple nous retrouvons les transactions[62] dans le domaine des bases de données, la logique temporelle des actions[40] dans le domaine de la vérification formelle, la mémoire transactionnelle[42] dans le domaine de la programmation concurrente, et les règles de réécriture de termes[32] dans le domaine de la synthèse matérielle. À la fin de cette thèse (chapitre 7), nous avons montré comment nous pouvons

intégrer plusieurs de ces travaux dans un cycle de développement complet d'un système sur puce.

- Le développement d'un environnement de modélisation et de simulation de systèmes sur puce basés sur le concept de transactions (voir Figure 1.1).

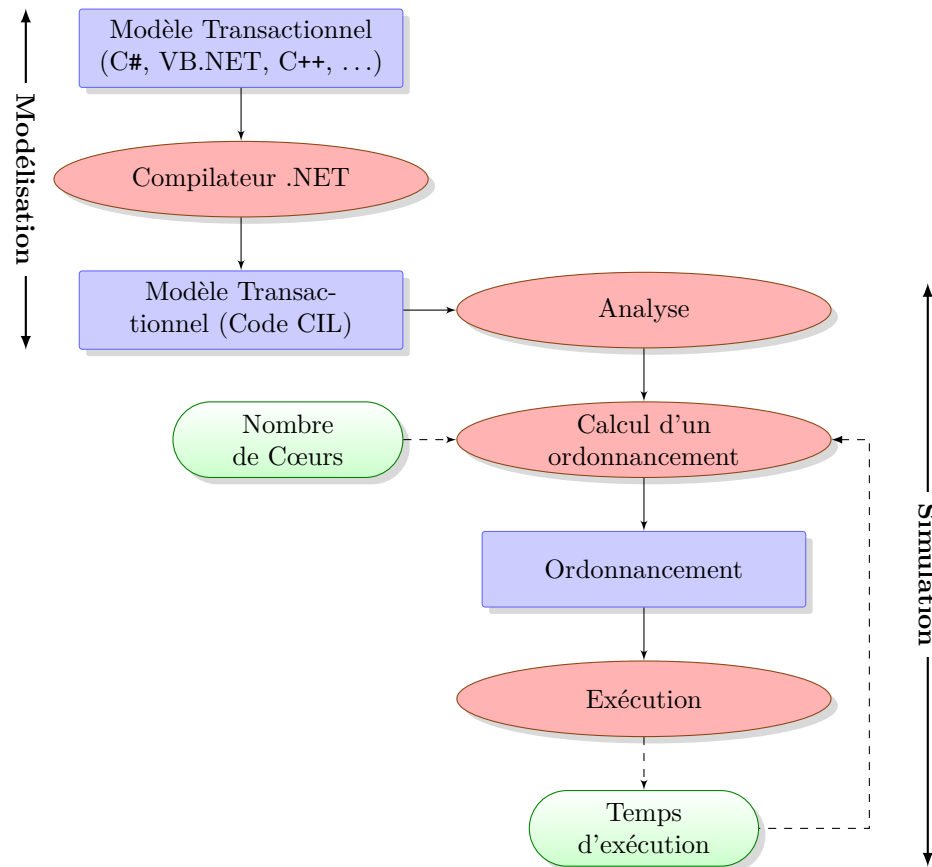


Figure 1.1 Environnement de Conception

Les contributions clés suite à ce travail sont :

- Séparation entre le modèle et le simulateur : Contrairement à SystemC, notre environnement assure une orthogonalité complète entre la modélisation et la simulation. En effet, un modèle SystemC compilé est un modèle exécutable qui contient l'implémentation du simulateur alors qu'un modèle transactionnel une fois compilé garde ses métadonnées, sa structure et son comportement, sans aucun détail sur comment il

va être simulé. Un des avantages de cette approche est de nous avoir permis d'utiliser le modèle avec différents simulateurs afin d'analyser plusieurs alternatives (chapitre 5). Vous pouvez consulter les travaux de [41] pour plus de détails sur les avantages que nous pouvons tirer grâce à cette séparation des aspects.

- **Analyseur intégré** : L'environnement de conception contient un analyseur intégré qui permet d'extraire du modèle transactionnel l'architecture et le comportement des différents composants ainsi que leurs interactions. Cet analyseur peut être aussi utilisé si nous voulons étendre l'environnement pour inclure d'autres outils pour la vérification formelle ou la synthèse matérielle. Ceci permet par exemple de transformer le modèle vers du code TLA⁺ ou Bluespec. La réalisation de cet analyseur est rendue moins compliquée grâce à l'introspection et la programmation par attributs que fournit la plateforme .NET. D'un autre côté, définir un analyseur pour SystemC était l'objet de plusieurs recherches intensives[46] vu la complexité du langage. En effet, les travaux dans [46] ont montré que, malgré le besoin croissant d'analyser des modèles SystemC suite à la complexité croissante des systèmes sur puce, les outils disponibles ne sont pas en mesure d'analyser des modèles SystemC arbitraires.
- **Environnement multi-langage** : Un modèle écrit en utilisant un langage de haut niveau tel que C# est compilé vers le langage intermédiaire commun CIL. Le simulateur utilise le langage CIL pour extraire toutes les informations nécessaires pour l'analyse du modèle transactionnel. En conséquence notre environnement de simulation est indépendant du langage et peut supporter plusieurs langages. Ceci constitue un atout majeur au cas où, par exemple, nous voulons décrire un système hétérogène avec des langages de descriptions spécifiques pour chaque sous-système. Une fois compilé, le système va avoir une représentation standard basée sur un modèle transactionnel décrit en langage CIL.

- Accélération de la Simulation : L'environnement de simulation tient compte de la structure du modèle transactionnel et de l'interaction entre les différentes transactions, du nombre de cœurs de la machine de simulation et d'une estimation du temps d'exécution afin de générer un ordonnancement décrivant comment les transactions doivent être ordonnancées et exécutées en parallèle. Nous avons testé cet environnement (chapitre 6) sur une étude de cas constituée d'un modèle de transmetteur selon la norme Wi-Fi 802.11a. Avec deux processeurs, nous avons obtenu une accélération d'environ 1.8, qui se rapproche d'une accélération optimum. Avec 8 processeurs, nous avons pu atteindre une accélération d'environ 4.6, qui est un résultat très prometteur.

1.5 Plan

Le chapitre 2 présente le flot de conception classique d'un système microélectronique et montre comment il ne répond plus aux exigences des systèmes sur puce. Ensuite il décrit la solution qui a été adoptée par l'industrie qui est basée sur la conception au niveau transaction (TLM) et le langage de description SystemC. Enfin, il détaille les limitations de ce langage pour satisfaire les nouveaux besoins dans la conception des SOC à cause de leur complexité qui ne cesse de s'accroître.

Le chapitre 3 introduit le concept de transaction que nous proposons comme méthodologie de conception des SOC pour remédier aux limitations de SystemC. Il montre le mérite de ce concept en tant que mécanisme de contrôle de concurrence. Il décrit différents travaux qui ont été faits autour des transactions et montre comment ces travaux peuvent être appliqués dans plusieurs aspects de la conception des systèmes sur puce tels que la modélisation, la vérification et la synthèse.

Le chapitre 4 présente une étude théorique sur les modèles transactionnels et leur environnement d'exécution. Il définit une sémantique formelle au modèle transactionnel et introduit plusieurs notions de validité afin de montrer comment une exécution concurrente des transactions doit se dérouler pour respecter la sémantique d'exécution du modèle transactionnel. Cette étude théorique constitue

la base pour le développement de l'environnement de simulation présenté au chapitre 5.

Le chapitre 5 décrit l'environnement de modélisation et de simulation qui a été réalisé en utilisant la plateforme .Net. Il montre comment un modèle transactionnel est représenté dans cette plateforme puis il décrit les étapes développées et les problèmes d'optimisation linéaire utilisés permettant de transformer un modèle transactionnel vers un modèle exécutable pour une simulation concurrente profitant d'une machine multicœur.

Le chapitre 6 détaille les différentes expérimentations qui ont été faites afin d'évaluer les performances d'une simulation parallèle en se basant sur un modèle transactionnel spécifiant un transmetteur selon la norme Wi-Fi 802.11a. Il étudie et compare les différentes alternatives déjà présentées dans le chapitre 5 puis il propose des optimisations afin de minimiser les surcoûts d'ordonnancement et augmenter le parallélisme entre les transactions.

Le chapitre 7 conclut cette thèse en rappelant les principales réalisations effectuées et les contributions apportées en matière de modélisation et simulation des systèmes sur puce. Ensuite il propose des perspectives en matière de vérification, raffinement et synthèse afin de compléter le cycle de conception de systèmes sur puce et pouvoir relever les défis présentés dans la section 1.1.

Conception des SOC¹

Au fil de l'évolution des systèmes microélectroniques industriels, les concepteurs ont été toujours forcés d'améliorer leur productivité afin d'exploiter pleinement la capacité croissante des circuits intégrés et ainsi réduire l'écart de productivité. L'approche utilisée visait toujours l'augmentation du niveau d'abstraction du modèle utilisé pour la description du système à implémenter. Ainsi, la représentation d'un système matériel est passée d'une représentation au niveau transistors à une représentation composée de portes logiques pour enfin arriver à une représentation basée sur les transferts de registres (RTL). Aujourd'hui, l'utilisation des modèles RTL comme base de départ dans le flot de conception des systèmes sur puce n'offre plus un niveau d'abstraction suffisant pour arriver à réduire l'écart entre la productivité de conception et la capacité des puces actuelles. Par conséquent, l'élaboration d'une nouvelle méthodologie de conception initiée à un niveau d'abstraction supérieur au niveau RTL, connue sous le nom de conception au niveau système, est devenue primordiale si l'on souhaite réduire cet écart.

Dans ce chapitre, nous donnons un aperçu du flot de conception classique et de ses limitations. Ensuite, une solution à ses limitations est présentée en introduisant la modélisation au niveau transaction, communément appelée TLM, et en montrant comment elle est adaptée au nouveau flot de conception. Finalement, nous introduisons SystemC qui est le langage de description adopté par l'industrie et grâce auquel le TLM est supporté. Nous présentons les inconvénients de SystemC qui nous ont poussés à adopter une nouvelle méthodologie qui est basée sur le concept des transactions (chapitre 3).

1. L'étude faite dans ce chapitre fait partie de nos publications[3]

2.1 Flot de conception classique et ses limitations

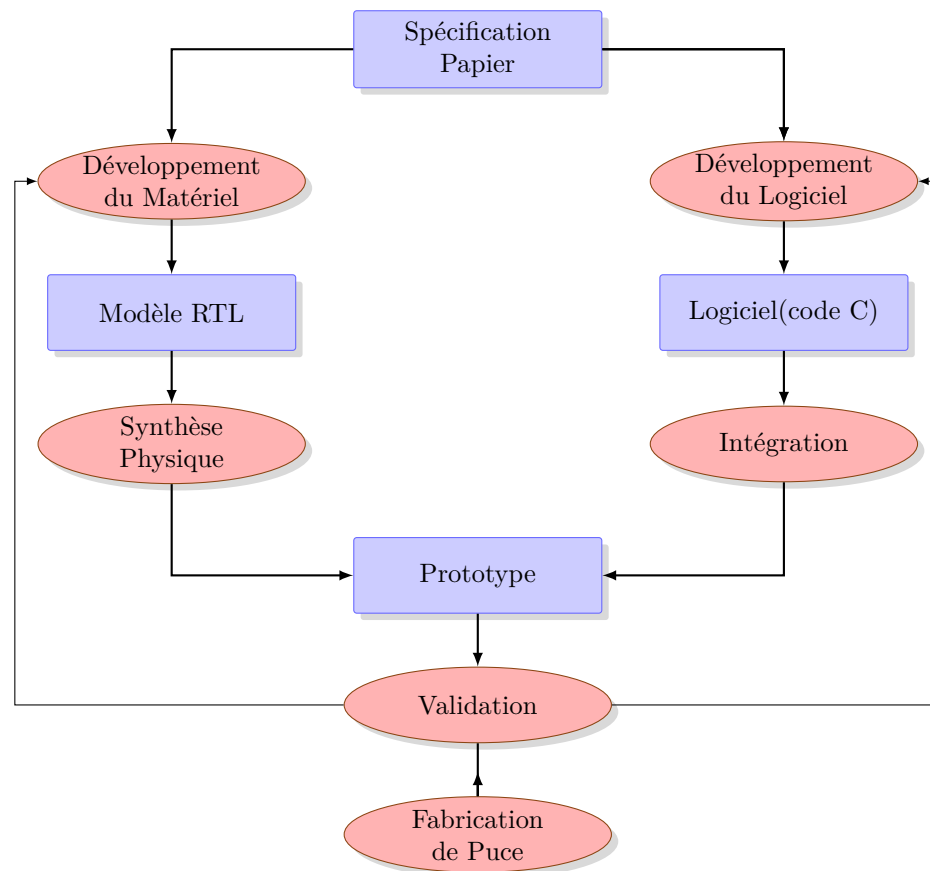


Figure 2.1 Flot de conception classique

Le flot de conception des systèmes sur puce se veut une méthodologie permettant de concevoir, vérifier et délivrer un SOC fonctionnellement correct tout en respectant des contraintes non fonctionnelles, telles que la latence, le débit et la puissance. Traditionnellement, ce flot de conception est divisé en deux activités séparées[23] (voir Figure 2.1) :

1. Développement du matériel
2. Développement du logiciel

Le développement du matériel débute par la définition de sa description RTL en utilisant un langage de description de matériel (HDL) tel que VHDL ou Verilog. Le modèle RTL développé est soumis à la vérification fonctionnelle par simulation afin de valider son comportement sur des jeux de tests. Ensuite vient la phase de la synthèse physique qui prend comme entrée le modèle RTL, puis génère une description en termes de portes logiques. Cette description est ensuite utilisée par des outils de placement et routage afin de générer la topologie du masque sur laquelle va se baser la fabrication d'un prototype physique du système.

L'autre activité dans le flot de conception est le développement du logiciel. Bien que le codage du logiciel puisse se faire en parallèle avec le matériel, la validation du comportement du logiciel ne peut se faire qu'une fois que le prototype physique a été généré. Si une erreur est détectée dans le matériel ou le logiciel, le flot de conception doit alors être répété jusqu'à l'obtention d'un système fonctionnellement correct et exhibant les performances requises. À ce moment le système est prêt à entrer dans le processus de fabrication.

Étant donné la complexité actuelle des systèmes sur puce et la nécessité de réduire le temps de leur mise en marché, le flot de conception classique n'est plus acceptable. Voici deux facteurs principaux expliquant l'obsolescence de ce flot de conception.

1. Validation du logiciel : Le développement du logiciel, spécialement le débogage et la validation, ne peut avoir lieu que si celui-ci s'exécute sur sa plateforme cible. Un prototype physique est utilisé comme point d'entrée pour la validation du logiciel. L'inconvénient majeur de cette approche est que le prototype n'est disponible qu'à la fin du cycle de développement de la plateforme matérielle cible. Ceci, non seulement accroît le temps de mise en marché du produit final, mais nécessite aussi la création d'un nouveau prototype si une erreur est détectée au niveau du matériel suite à l'exécution du logiciel. Ceci est un processus très coûteux nécessitant des équipements chers.

La covérification matériel/logiciel[16] peut aussi être utilisée afin de simuler

le logiciel avec le modèle RTL de la plateforme. Cependant, il faut encore attendre assez longtemps avant que le modèle RTL soit prêt. En plus, la covérification est exécutée à la vitesse faible du RTL. Par conséquent, seulement de petites portions du code logiciel peuvent être simulées, par exemple, pour déboguer le pilote logiciel d'un périphérique matériel.

Grâce à la capacité croissante des circuits intégrés, un système sur puce complexe peut contenir une plateforme multicœur ayant des processeurs multiples. Ainsi, le logiciel est utilisé pour réaliser une grande partie de la fonction globale d'un SOC, vu que son temps de développement est très inférieur à celui du matériel exécutant la même fonction. En plus, le logiciel fournit une plus grande flexibilité face à l'évolution du système, que ce soit en cours de développement ou après la mise sur le marché. Par exemple, si une nouvelle version d'une norme doit être prise en compte, il est possible de fournir une mise à jour du logiciel que l'utilisateur peut télécharger. Toutefois, afin de pouvoir profiter de ces avantages que le logiciel offre par rapport au matériel, il faut pouvoir commencer à développer et vérifier une forte proportion du logiciel beaucoup plus tôt dans le cycle de développement du système sans devoir attendre la plateforme RTL ou le prototype. En plus, il importe que le logiciel puisse être simulé à une vitesse supérieure que celle obtenue par la méthode de covérification.

- 2. Exploration architecturale :** Les systèmes sur puce contiennent aujourd'hui plusieurs IP, tels qu'un ou plusieurs processeurs, DSP, mémoires, périphériques, etc. Ces IP communiquent à travers un ou plusieurs bus. La configuration de ces bus ainsi que le choix des protocoles influent considérablement sur les performances [3]. Les architectures à bus partagés, tels que OCP, AMBA et CoreConnect, constituent des choix populaires pour les systèmes sur puce. Ils permettent un espace d'exploration large vu qu'ils peuvent être configurés de différentes manières. Ainsi, le concepteur du système peut explorer différents protocoles de communication et plusieurs configurations afin de faire le choix d'une architecture à adopter en vue de répondre aux

exigences de performance. L'utilisation d'un modèle RTL pour explorer plusieurs types d'architectures est impraticable, dû à ses temps de simulation trop longs.

Un autre point auquel le flot de conception classique ne peut pas répondre est l'exploration architecturale que l'on effectue pour décider des fonctionnalités qui seront implémentées en logiciel et celles qui seront implémentées en matériel. Le matériel permet généralement d'avoir de meilleures performances comparées au logiciel. Le processus de développement du matériel est toutefois plus long et plus coûteux que celui du logiciel. Il faut cependant rappeler que les systèmes sur puce sont soumis à des contraintes de temps réel que le logiciel ne pourrait satisfaire à lui seul. Ainsi, les fonctionnalités ciblées par des contraintes de performance sont implémentées en matériel alors que les autres fonctionnalités sont écrites en logiciel. On appelle partitionnement matériel/logiciel[10] le problème délicat qui consiste à trouver la bonne architecture, c'est-à-dire celle possédant les performances requises et développée au moindre coût.

2.2 Modélisation au niveau transaction (TLM)

Afin de faire face à la complexité croissante des systèmes sur puce et répondre aux exigences présentées plus haut, le point d'entrée du flot de conception doit être élevé à un niveau plus haut que RTL, d'où vient le terme « conception au niveau système » (SLD : System Level Design). Durant ces 10 dernières années, la conception au niveau système a eu un grand intérêt pour une nouvelle méthodologie de conception, connue sous le nom de modélisation au niveau transaction (TLM : Transaction-Level Modeling)[14]. Le principe fondamental de TLM est de pouvoir modéliser seulement le niveau de détail qui est nécessaire pour une tâche particulière en cours du cycle de développement. Ce niveau de détail est caractérisé par son degré de précision par rapport à une implémentation réelle du système. Plusieurs aspects permettent de déterminer le niveau de précision d'un modèle[25]. Voici les deux aspects principaux utilisés dans TLM.

- **Aspect structurel** : C'est le degré auquel le modèle reflète la structure concrète de l'implémentation. Ceci dépend de plusieurs facteurs. Par exemple :
 - Si le partitionnement matériel et logiciel est déjà défini.
 - Si les modules matériels et logiciels de l'implémentation sont représentés.
 - Pour les modèles matériels, si les signaux et les broches de l'implémentation sont reflétés dans le modèle.
 - Pour les modèles logiciels, si la communication entre tâches a été raffinée au niveau des mécanismes de communication fournis par le système d'exploitation temps réel (RTOS) utilisé dans l'implémentation.
 - Pour la communication, si le protocole de communication de l'implémentation est spécifié.

- **Aspect temporel** : C'est le degré auquel le modèle représente le temps de calcul réel de l'implémentation. Trois niveaux de temps sont généralement distingués sur un modèle particulier :
 - *Modèle Un-timed* : C'est un modèle où la notion de temps n'existe pas.
 - *Modèle Approximate-timed* : C'est un modèle qui possède la notion de temps afin de représenter une estimation du temps d'exécution réel de l'implémentation.
 - *Modèle Cycle-timed* : C'est le modèle le plus précis qui détermine exactement le temps d'exécution réel de l'implémentation en nombre de cycles d'horloges.

La méthodologie basée sur TLM n'est pas unique et plusieurs modèles transactionnels ont été définis[23, 14, 25, 51] selon leurs degrés de précision temporelle et structurelle. Dans ce qui suit, nous allons donner un aperçu de TLM en identifiant les concepts communs qui l'entourent.

Le premier concept sur lequel est basé le TLM est le suivant : la communication entre les composants qui constituent un système sur puce et la fonctionnalité

de ces composants peuvent être développées et raffinées indépendamment. Ainsi, dans un modèle TLM, les communications telles que FIFO ou bus sont modélisées par des canaux et les composants tels que contrôleurs, processeurs, DSP, sont modélisés par des modules. Les modules communiquent entre eux en faisant appel à des fonctions d'interfaces exposées par les canaux. Ces appels d'interface sont appelés transactions, d'où vient le terme TLM. La modélisation de la fonctionnalité des composants ainsi que la communication entre eux passe par trois niveaux d'abstraction :

1. **Niveau fonctionnel** : C'est le plus haut niveau d'abstraction. Il décrit la fonctionnalité du système sans donner aucun détail d'implémentation temporel ou structurel. Par conséquent, un tel modèle n'a aucune notion de composant matériel ou logiciel et l'architecture du SOC n'est pas encore représentée.
2. **Niveau d'architecture** : Une fois que la fonctionnalité du système est définie, la phase suivante consiste à spécifier l'architecture de la plateforme. Ce niveau doit avoir un degré de précision structurelle suffisant afin de pouvoir commencer à développer et valider le logiciel.
3. **Niveau microarchitecture** : Ce niveau capture toute l'information qui permet une simulation niveau Cycle-timed. Le matériel est spécifié par sa description RTL alors que le logiciel est spécifié par le jeu d'instructions du processeur cible. Ce modèle est le point d'entrée des outils de synthèse physique.

La Figure 2.2 illustre les différents modèles qui peuvent être utilisés durant le cycle de développement en considérant les trois niveaux d'abstraction cités plus haut, ainsi que la séparation entre communication et fonctionnalité.

Les deux modèles extrêmes dans le flot de conception sont le modèle fonctionnel et le modèle RTL.

- **Modèle fonctionnel** : appelé aussi modèle algorithmique[23] ou modèle de spécification[14]. C'est une spécification exécutable décrivant la fonctionna-

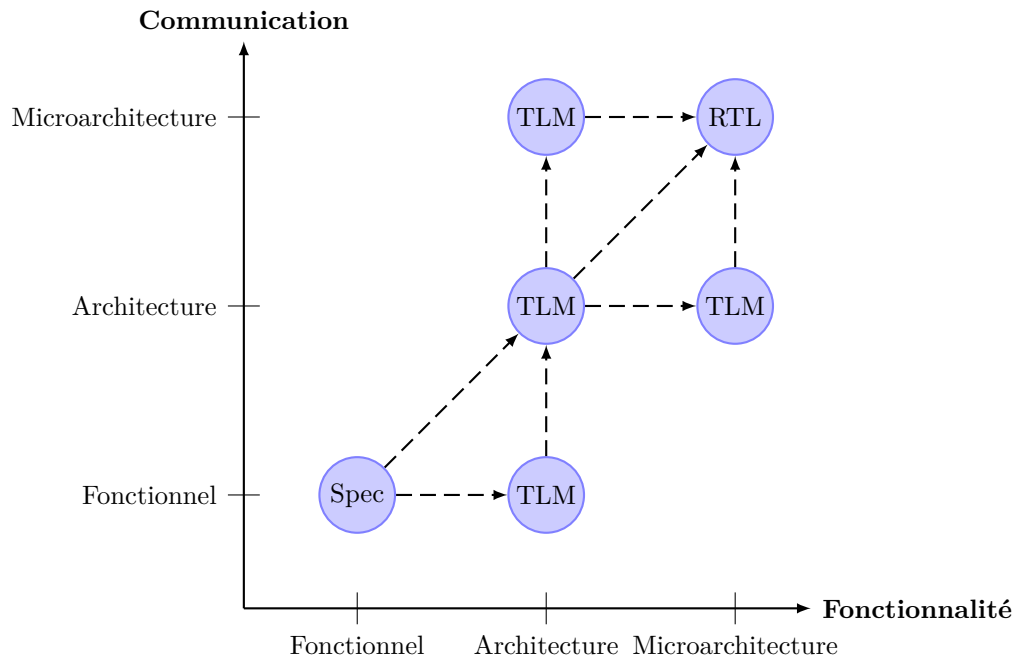


Figure 2.2 Modèles TLM

lité du système sans détail de l'architecture. Ce modèle est souvent décrit par un ensemble de processus qui communiquent entre eux à travers des variables partagées ou par des canaux de passage de messages tels que des FIFO. Par exemple, un système orienté traitement numérique de signal, va avoir un modèle de flux de données simulable, où les traitements sont représentés par des processus qui communiquent à travers des FIFO.

- **Modèle RTL** : Un modèle RTL décrit le comportement d'un circuit synchrone en spécifiant deux sortes d'éléments : registres et logique combinatoire. Les registres sont les composants qui contiennent les données. La logique combinatoire spécifie toutes les fonctions logiques d'un circuit. Ces fonctions prennent en entrée les données stockées dans un ou plusieurs registres et calculent les valeurs de sortie vers ces registres. La mise à jour des registres est synchronisée et n'aura lieu qu'au moment où un signal horloge (ou plusieurs) est activé. Ce modèle décrit toute la microarchitecture au cycle près.

Le flot de conception entre le modèle fonctionnel et le modèle RTL peut passer par plusieurs modèles intermédiaires. Le point commun entre ces modèles intermédiaires est que l'architecture de la plateforme cible est déjà spécifiée avec des degrés de précision structurels différents. Tous ces modèles sont considérés comme étant des modèles TLM[14]. D'un autre côté, ils peuvent refléter certains degrés de précision temporelle qui, par conséquent, vont engendrer d'autres niveaux d'abstraction. Ces modèles TLM ne sont pas nécessairement tous utilisés durant le cycle de développement. Ceci dépend considérablement des techniques utilisées et de la complexité du système à développer[14]. Dans ce qui suit, nous allons donner une vue globale sur l'utilité des modèles TLM, et montrer comment ils répondent aux besoins cités dans la section précédente.

Développement du logiciel au tout début

Dès que la spécification du système a été complétée, le développement de la plateforme TLM commence. Ainsi, la plateforme cible sera disponible pour le développement du logiciel très tôt dans le cycle de développement. Ceci permet d'avoir une méthodologie de conception matériel/logiciel concurrente où le développement du logiciel se fait en parallèle avec le matériel. En effet, les développeurs de logiciels regardent la plateforme TLM comme le modèle de référence pour exécuter et valider leurs codes logiciels, alors que les développeurs du matériel la considèrent comme le modèle de départ pour entamer le développement de la plateforme RTL. Vis-à-vis du développement logiciel, la plateforme TLM est utilisée pour les deux objectifs suivants :

- la validation de la fonctionnalité logicielle en utilisant une plateforme TLM non temporelle. Ceci correspond au modèle d'architecture TLM de la Figure 2 où l'architecture est décrite au niveau fonctionnalité et communication. Les détails sur la microarchitecture sont ignorés. Par exemple, les détails sur le protocole de bus et la microarchitecture pipeline, n'influent pas sur la fonctionnalité et ne sont pas précisés à ce niveau d'abstraction. Ceci a pour avantage d'avoir une rapidité de simulation suffisante pour pouvoir dérouler le logiciel et le valider.

- L'optimisation de certaines fonctionnalités logicielles soumises à des contraintes temporelles en utilisant un modèle TLM temporisé. Dans ce cas, le modèle d'architecture va être annoté par des délais approximatifs afin de pouvoir estimer le temps d'exécution de la plateforme cible. Il est parfois nécessaire d'avoir un temps d'exécution précis. Ainsi, certaines fonctionnalités au niveau des composants ou de la communication vont être raffinées au niveau microarchitecture afin d'obtenir un modèle cycle-timed. Ces modèles temporels sont aussi utilisés afin d'estimer les performances et faire de l'exploration architecturale.

Exploration architecturale

Durant les premières phases du cycle de développement, il est nécessaire de définir l'architecture qui convient au mieux aux besoins et qui satisfait les contraintes de performance exigées. En raison de sa rapidité de simulation, TLM donne la possibilité d'analyser plusieurs architectures du système, une fois la spécification fonctionnelle du système est complétée. Ainsi, le concepteur a les moyens de décider sur le bon partitionnement matériel/logiciel, la topologie de la communication, les types des bus et des processeurs à utiliser, etc. Afin de pouvoir estimer les performances du système, le modèle TLM doit inclure des informations sur le temps d'exécution. Ces informations de temps peuvent être extraites à la suite de conduites de tests de performance ou à partir des modèles RTL de certains IP déjà développés et qui vont être réutilisés dans le système courant [51].

Dans cette section, nous avons présenté les concepts fondamentaux entourant la méthodologie de conception basée sur TLM. Cependant, il est nécessaire d'avoir les outils et les techniques nécessaires afin de mettre en pratique cette méthodologie et ainsi profiter de tous les avantages qu'elle peut apporter. Aujourd'hui, le standard utilisé dans la conception des systèmes sur puce en se basant sur la méthodologie TLM est SystemC. Dans la prochaine section, nous allons donner un aperçu de SystemC et de ses limitations qui constituent un obstacle pour répondre aux nouveaux besoins des systèmes sur puce.

2.3 SystemC et ses limitations

Plusieurs systèmes et langages ont émergé afin de résoudre plusieurs aspects de la conception des systèmes sur puce. Par exemple, les langages de description de matériel tels que VHDL et Verilog sont utilisés pour simuler et synthétiser les circuits numériques. C/C++ est le langage principal utilisé pour développer les logiciels embarqués. PSL et SVA sont utilisés pour la vérification fonctionnelle des modèles RTL. SystemVerilog, une extension de Verilog, élève le niveau d'abstraction pour la description du matériel en ajoutant des structures orientées objets en permettant la synthèse comportementale. Bluespec et Esterel permettent la génération d'un modèle RTL à partir d'une description matérielle à un plus haut niveau d'abstraction. Matlab est souvent utilisé pour la spécification des besoins et le développement des algorithmes de traitement de messages.

Chacun de ces langages a un rôle particulier au cours de cycle de développement. Cependant, la complexité croissante des systèmes sur puce nécessite entre autres :

- La disponibilité d'un système de modélisation fiable qui permet d'assister les concepteurs durant tout le cycle de développement en commençant à un niveau d'abstraction système.
- Pouvoir échanger entre compagnies des IP écrits à un niveau d'abstraction système. Pour cela il faut que les compagnies adoptent le même langage de modélisation.

Grâce à l'organisme OSCI (Open SystemC Initiative), composé de plusieurs compagnies industrielles, le langage SystemC a pris naissance en 1999, et en 2005 il est devenu une norme IEEE. SystemC n'est pas la meilleure solution pour tous les aspects de conception, cependant il unit plusieurs caractéristiques qui manquent dans d'autres langages. Voici quelques unes.

- La spécification et la modélisation à plusieurs niveaux d'abstraction, commençant par un modèle purement fonctionnel jusqu'à un modèle RTL.

- Des mécanismes qui séparent la fonctionnalité du système de sa communication, permettant ainsi l'adoption de la méthodologie TLM.
- La notion de temps afin de pouvoir estimer les performances et explorer plusieurs alternatives.
- Avoir un seul langage de conception pour modéliser le matériel et le logiciel afin de faciliter le codesign matériel/logiciel et d'accélérer la simulation.

Nous commençons par introduire la structure d'un modèle SystemC et son modèle d'exécution évènementiel. Ensuite, nous présentons les limitations de SystemC qui sont dues principalement à ce modèle de calcul.

2.3.1 Vue d'ensemble

SystemC utilise une approche en couches qui permet une grande flexibilité afin d'introduire de nouvelles structures qui partagent le même noyau de simulation. Ce noyau de simulation est la couche de base de SystemC. Il est basé sur un ordonnanceur évènementiel qui travaille avec les processus et les évènements d'une manière abstraite. Il contrôle les évènements et l'ordonnancement des processus sans avoir connaissance de ce que les évènements représentent ou de ce que font les processus. Ensuite vient la couche des éléments structurels qui permet de spécifier la structure des modèles. Un modèle SystemC est composé d'un ou plusieurs modules. Ces modules communiquent avec d'autres modules à travers des ports reliés à des canaux de communications. Les ports exposent des interfaces qui sont implémentées par les canaux. De cette manière, il est possible d'analyser plusieurs types de canaux implémentant la même interface sans avoir à modifier les modules communicants. Un module peut avoir un ou plusieurs processus qui décrivent sa fonctionnalité. La synchronisation entre processus se fait par notification et attente d'évènements. Au-dessus de cette couche sont définis des canaux standards ainsi que des canaux spécifiques à des modèles de calcul. On retrouve aussi la bibliothèque TLM et la bibliothèque de vérification. SystemC définit un ensemble de types de données qui sont utiles pour les modèles matériels et certains types

d'applications logicielles. À titre d'exemple, il y a le type fixed-point pour les applications DSP (Data Signal Processing).

La Figure 2.3 représente les différentes couches de SystemC. La couche de fond montre que SystemC n'est pas un langage à lui seul, mais il s'agit d'une bibliothèque fondée entièrement en C++. C'est-à-dire que n'importe quel programme écrit en SystemC peut être compilé avec un compilateur C++ afin de générer un programme exécutable.

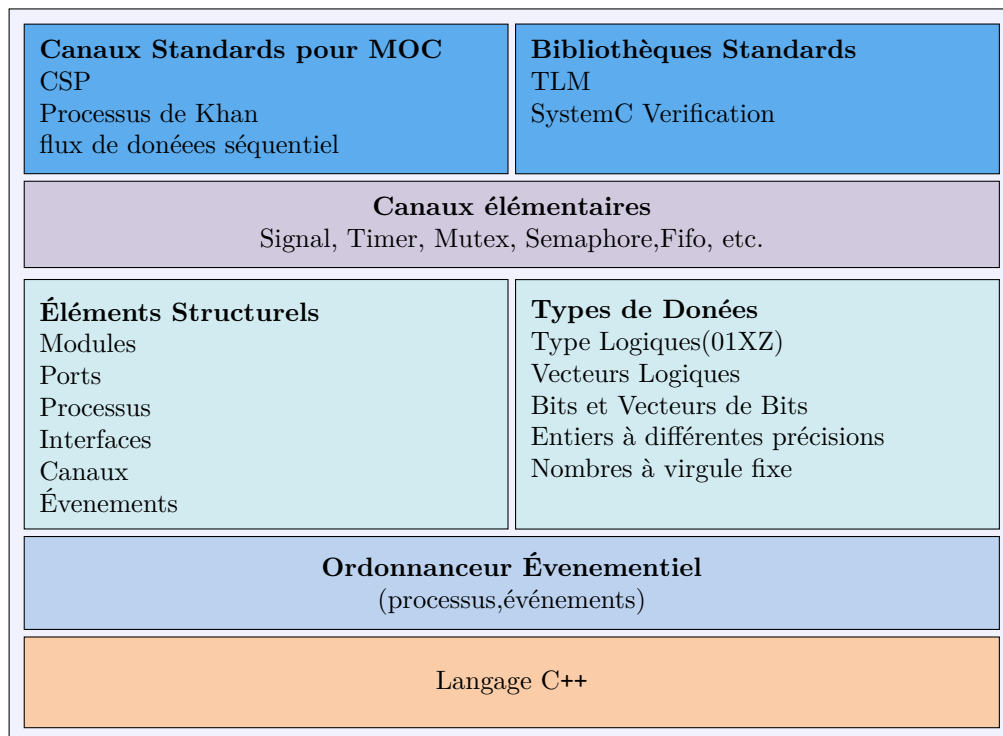


Figure 2.3 Architecture du langage SystemC

L'algorithme d'ordonnancement de SystemC se base sur un modèle d'exécution événementiel. Il distingue trois types de notification d'évènements : la notification immédiate, la notification delta et la notification temporelle. La coordination et la synchronisation entre les processus suite à ces trois types de notification sont gérées par l'ordonnanceur de SystemC dont les trois phases d'exécution sont représentées dans la Figure 2.4.

Phase d'initialisation : Pendant la phase d'initialisation, chaque processus est

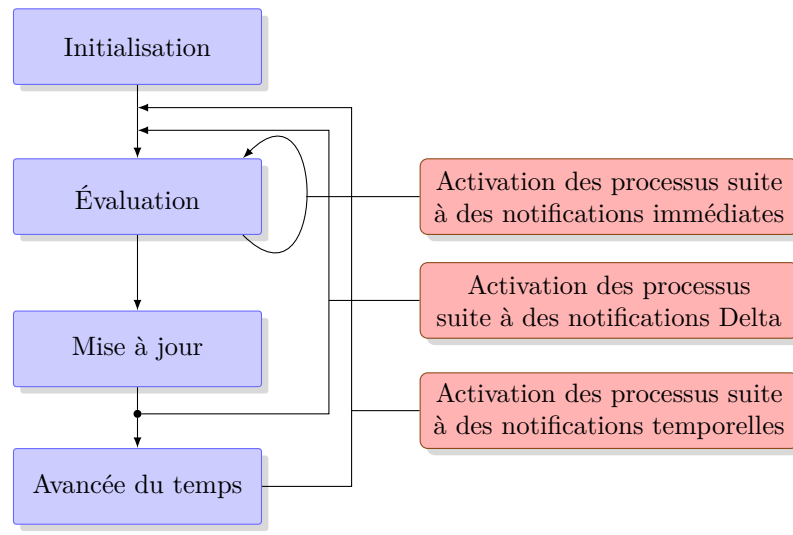


Figure 2.4 Modèle d'exécution de SystemC

exécuté une fois jusqu'à sa suspension. Un processus est suspendu lorsqu'il appelle la fonction `wait`, indiquant ainsi qu'il passe en mode attente en passant la main à l'ordonnanceur. Il y a trois types d'attentes : attente de la notification d'un événement, attente du prochain cycle delta, ou attente temporelle définie par une durée de temps discret.

Phase d'évaluation : Après la phase d'initialisation, l'ordonnanceur commence la phase d'évaluation en exécutant séquentiellement dans un ordre indéterminé la liste des processus prêts à être exécutés. Durant cette phase, un processus pourrait générer une notification d'évènement immédiat. Ainsi, d'autres processus sensibles à cet évènement sont ajoutés à la liste des processus prêts à être exécutés. Ces nouveaux processus seront exécutés dans la même phase d'évaluation. Une fois que la liste des processus prêts à être exécutés est vide, l'ordonnanceur passe à la phase de mise à jour.

Phase de mise à jour : La phase de mise à jour permet d'implémenter la sémantique de demander-actualiser (`request-update`) permettant d'ajourner l'actualisation de l'état du système une fois que la phase d'évaluation est terminée. La combinaison de la phase d'évaluation et de la phase de mise

à jour forme un delta-cycle. Ainsi, à la fin du delta-cycle, l'ordonnanceur vérifie s'il existe des notifications delta en instance. Si c'est le cas alors, l'ordonnanceur enregistre tous les processus en attente de l'occurrence de ces événements dans une liste de processus prêts à être exécutés puis recommence un nouveau delta-cycle en exécutant encore une fois la phase d'évaluation.

Avancée du temps : Quand il n'existe plus de notification delta, l'ordonnanceur avance le temps à la plus proche notification temporelle (dû à une attente temporelle), détermine les processus à être exécutés durant cet instant de temps, puis recommence l'exécution des trois phases d'ordonnement correspondant à un nouveau temps de simulation.

2.3.2 Limitations de SystemC

Dans ce que suit nous présentons les inconvénients majeurs de SystemC selon plusieurs aspects de conception.

2.3.2.1 Modélisation

SystemC ne rend pas la tâche de modélisation très différente de celle d'un langage de description du matériel (HDL) tel que VHDL ou Verilog. Ceci est dû au fait que la sémantique du langage ne diffère pas de celle d'un langage HDL qui est basé sur un modèle de calcul événementiel. En effet, à part la notification immédiate qui a été ajoutée à la version 2.0 de SystemC, les deux types de langages se basent sur la même notion de delta-cycle et des notifications d'événement pour la synchronisation entre processus. Par conséquent, les mêmes difficultés rencontrées, au niveau des langages HDL, pour gérer les accès concurrents à des ressources partagées sont aussi présentes dans SystemC[52]. D'un autre côté, même au niveau TLM et plus haut, SystemC utilise le même modèle de calcul augmenté des notifications immédiates. Comme nous allons voir dans la section 3.1, avec la complexité croissante des systèmes qui nécessite de plus en plus de processus parallèles, l'utilisation de ces mécanismes de synchronisation devient de plus en plus complexe et le risque d'erreurs est élevé.

Un autre problème qui peut survenir lors de la modélisation d'un système SystemC est l'interblocage dû à l'ordre d'exécution des processus. En effet, lors de la phase d'évaluation, l'ordre des processus est arbitraire. Cependant pour une implémentation particulière de SystemC et pour un même stimulus l'ordre des processus est toujours le même. Par conséquent, la simulation de certains modèles génère des résultats corrects dans une implémentation particulière de SystemC, alors que le modèle exhibe certaines défaillances fonctionnelles. L'exemple 2.1 illustre ce problème dû à l'emploi des notifications immédiates.

Exemple 2.1 *Interblocage dû aux notifications immédiates*

```
writer = new producer("Writer") // line 1
reader = new reader("Reader") // line 2

void read(){
    while(true)
    {
        wait(write_event);
        ConsumeValue(value);
        read_event.notify();
    }
}

void write() {
    while(true)
    {
        value = ComputeNewValue();
        write_event.notify();
        wait(read_event);
    }
}
```

Si l'ordonnanceur exécute le processus `writer` en premier, alors la notification immédiate de l'événement `write_event` est perdue puisqu'aucun processus n'est en attente de cet événement lors de son occurrence. Ceci mène à un interblocage où le modèle est suspendu indéfiniment. Cependant, si le processus « `reader` » s'exécute en premier, il sera suspendu en attente de la notification de l'événement `write_event` et ainsi, quand le processus `writer` s'exécute, l'événement `write_event` est notifié, permettant au processus `reader` de continuer son exécution et le cycle write-read continuera indéfiniment. Remarquer qu'il est possible d'influencer l'ordre d'exécution des processus en interchangeant les déclarations des processus `writer` et `reader` dans les lignes 1 et 2 du présent exemple. Ce genre d'erreur dû au non-déterminisme dans l'ordre

d'exécution des processus, pourtant fixe pour une implémentation particulière de SystemC, peut mener à une implémentation incorrecte du modèle dont la simulation a donné des résultats corrects.



Pour éliminer le comportement indésirable induit par le non-déterminisme qui est dû à l'utilisation des notifications immédiates, la plupart des canaux standards de SystemC, incluant `sc_fifo`, se basent sur la sémantique request-update et le concept de delta-cycle expliqués plus haut[34]. En effet, si nous remplaçons les deux notifications immédiates, dans l'exemple 2.1, par des notifications delta, `notify(SC_ZERO_TIME)` (voir Exemple 2.2), nous obtenons un comportement déterministe indépendant de l'ordre d'exécution des processus. Cependant, une telle synchronisation implicite entre processus imposée dans chaque delta-cycle est très restrictive comparée à une synchronisation effective entre processus parallèles dans une application logicielle multitâches ou dans un circuit asynchrone[37].

Exemple 2.2 *Interblocage résolu par notifications delta-cycle*

```

writer = new producer("Writer") // line 1
reader = new reader("Reader") // line 2

void read(){
    while(true)
    {
        wait(write_event);
        ConsumeValue(value);
        read_event.
        notify(SC_ZERO_TIME);
    }
}

void write() {
    while(true)
    {
        value = ComputeNewValue();
        write_event.
        notify(SC_ZERO_TIME);
        wait(read_event);
    }
}

```



D'un autre côté, un comportement non déterministe n'est pas toujours dû à des

erreurs de synchronisation, mais il peut être spécifié intentionnellement. En effet, la modélisation d'un système à un niveau très haut d'abstraction est par nature non déterministe puisque plusieurs détails d'implémentation sont abstraits. Par exemple, prenons le cas où plusieurs maîtres accèdent à un bus concurremment. Au point de vue de l'implémentation, cet accès concurrent est géré par un arbitre et des priorités d'accès. Cependant, au niveau système, l'ordre de priorité n'est pas encore spécifié. Autrement dit, quel que soit l'ordre d'exécution, il faut que la fonctionnalité du système soit correcte. Permettre un ordre aléatoire en cours de simulation ne résout pas le problème. Les travaux dans [29] et [38] adressent ce problème en générant plusieurs ordonnancements possibles en cours de simulation. Des techniques de réduction d'ordre partiel sont utilisés afin d'éliminer les ordonnancements qui ont les mêmes effets sur le comportement du système. Ces travaux permettent d'avoir une meilleure couverture de test, cependant l'impact sur le temps de simulation est considérable.

2.3.2.2 Vérification par raffinement et sémantique formelle

Bien que SystemC permette la modélisation à plusieurs niveaux d'abstraction système, le flot pour passer d'un niveau d'abstraction à un autre est géré dans la plupart des cas manuellement. Par conséquent, à chaque niveau d'abstraction, il faut revérifier la fonctionnalité du système afin d'assurer qu'elle est conforme aux spécifications initiales. D'un autre côté, les techniques utilisées pour la vérification fonctionnelle sont essentiellement basées sur la simulation. Cependant, l'inconvénient majeur de la simulation est qu'elle permet seulement une faible couverture même avec un temps de simulation extrêmement long. Comme il a été indiqué par ITRS, les défis actuels sont centrés autour de comment rendre les techniques de vérification formelle plus fiables et plus contrôlables. En dépit, des activités intensives dans le développement des techniques de vérification pour le logiciel et le matériel, étendre ces techniques à SystemC est un considérable défi[60]. En effet, plusieurs travaux ont essayé de donner une sémantique formelle à SystemC en le transformant à un langage formel tel que ASM[26], Promela[59], SMV[56] afin d'appliquer les techniques de «*Model Checking*». Toutefois, ces travaux traitent

seulement un sous-ensemble de SystemC. Cette difficulté est due essentiellement à la nature orientée objet de SystemC et à son modèle de calcul complexe.

2.3.2.3 Vitesse de simulation

Grâce à la méthodologie basée sur TLM, la vitesse de simulation est considérablement améliorée, permettant ainsi d'explorer plusieurs architectures et de pouvoir valider le logiciel[23]. Cependant, cette vitesse doit être maintenue au même taux de complexité que les systèmes sur puce. La disponibilité croissante des machines multicœurs est la tendance actuelle afin d'accélérer le temps de simulation. Toutefois, la complexité du langage SystemC rend la parallélisation de la simulation assez difficile.

En effet, le modèle de concurrence de l'ordonnanceur de SystemC repose sur la sémantique de coroutine. Voici la description de cette sémantique comme décrite dans [34] :

“Since process instances execute without interruption, only a single process instance can be running at any one time, and no other process instance can execute until the currently executing process instance has yielded control to the kernel. A process shall not pre-empt or interrupt the execution of another process. This is known as co-routine semantics or co-operative multitasking”.

Ainsi, chaque processus s'exécute sans interruption jusqu'à ce qu'il cède la main à un autre processus, à la suite de l'appel à la fonction wait, à partir de ce moment il reste suspendu jusqu'à sa prochaine exécution. Par conséquent, une manière directe de respecter cette sémantique est d'exécuter les processus séquentiellement. En fait, c'est cette exécution séquentielle qui est implémentée dans le simulateur de référence de SystemC. Il est toujours possible de concevoir une implémentation du simulateur qui profite d'une machine multicœur en exécutant les processus d'une manière concurrente à condition que le comportement apparaisse identique à la sémantique de coroutine. En d'autres termes, une analyse complexe des processus et de leurs dépendances est nécessaire pour une simulation multicœur d'un modèle SystemC. Plusieurs travaux[20, 22, 55, 47] ont essayé de paralléliser le simulateur de SystemC. Cependant vu la complexité du langage, seulement un

sous ensemble de SystemC est traité où le concepteur doit se soumettre à des contraintes de modélisation. En plus, les résultats de simulation ne sont pas toujours satisfaisants. Par exemple, les travaux dans [20, 22] ont essayé de passer d'un modèle de coroutine à un modèle préemptif où plusieurs processus peuvent s'exécuter en parallèle. Malheureusement, ceci ne donne pas toujours de meilleure performance, et souvent la vitesse de simulation se dégrade par rapport à une exécution séquentielle monoprocesseur. Ceci est dû essentiellement au fait que les processus décrivant la fonctionnalité du modèle sont partitionnés selon l'architecture de plateforme cible[55]. Prendre ces processus tels qu'ils sont, puis les exécuter concurremment peut engendrer des dégradations de performance à cause du surcoût qui est dû aux changements de contexte et à la synchronisation entre ces processus. Ce qu'il faut est de pouvoir transformer le modèle initial en un modèle fonctionnellement équivalent ciblant la plateforme de simulation. C'est de cette façon que nous avons procédé pour paralléliser l'exécution et accélérer le temps de simulation d'un modèle transactionnel (voir chapitre 5).

Malgré les travaux intensifs effectués pour remédier aux limitations citées plus haut, aussi bien académiques qu'industriels, les résultats obtenus sont toujours insatisfaisants. Nous pensons que ceci est dû essentiellement au modèle d'exécution complexe de SystemC et à son manque de sémantique formelle. Ainsi, l'approche que nous envisageons afin de résoudre ces questions est de se baser sur un nouveau modèle de calcul, moins complexe, assez expressif afin de représenter des systèmes à plusieurs niveaux d'abstraction, et ayant une sémantique formelle. Ce modèle d'exécution se base sur le concept de transaction qui fait l'objet du prochain chapitre.

Les Transactions¹

Le concept des transactions constitue un mécanisme puissant afin de gérer la concurrence entre plusieurs composants parallèles. Une transaction permet d’encapsuler un ensemble d’opérations en une seule opération indivisible. Ceci permet de décrire la fonctionnalité d’un système en plusieurs transactions concurrentes en raisonnant sur le comportement de chaque action en isolation par rapport aux autres. Par ailleurs, le concepteur ne se soucie plus de la gestion manuelle des ressources partagées et de la coordination entre les composants parallèles qui y accèdent.

Le modèle de calcul basé sur les transactions a été largement étudié aussi bien dans le domaine matériel que dans le domaine logiciel. Dans ce qui suit, nous présentons l’état de l’art des différentes recherches faites autour des transactions qui peuvent être intégrées dans le processus de conception des systèmes sur puce sur plusieurs aspects tels que modélisation, synthèse et vérification.

3.1 Mémoire transactionnelle (TM[42])

En premier lieu, nous allons introduire la notion de mémoire transactionnelle ainsi que ses avantages en tant que mécanisme de contrôle de concurrence. Ensuite, nous énumérons plusieurs implémentations logicielles de ce concept dans le domaine de la programmation concurrente.

3.1.1 Avantages de TM

Ces dix dernières années marquent le début de transition de la programmation séquentielle à la programmation parallèle, dû à la disponibilité croissante des

1. À la suite de l’étude faite dans ce chapitre, nous avons publié un chapitre de livre[4]

processeurs multicœurs que nous retrouvons dans la plupart des serveurs, des ordinateurs personnels et des systèmes sur puces[42].

La conception d'un programme parallèle comprend toute la difficulté de la programmation séquentielle, en plus, elle introduit le problème de coordonner l'interaction entre les différentes tâches qui s'exécutent en parallèle. Aujourd'hui la plupart des programmes parallèles utilisent des mécanismes de bas niveau pour gérer cette interaction. Ces mécanismes consistent en plusieurs *threads*, représentants des processeurs abstraits, et en constructions de synchronisation, telles que des sémaphores, des mutex, des verrous ou moniteurs, qui coordonnent l'exécution de ces *threads*. Cependant, l'expérience a montré que les programmes parallèles écrits avec ces mécanismes de synchronisation sont difficiles à concevoir, debugger, maintenir et la plupart du temps ne sont pas assez performants[42]. Ces difficultés de concevoir un programme parallèle ont mené à trouver une meilleure abstraction pour écrire ce genre de programmes. Cette abstraction est connue sous le nom de mémoire transactionnelle (TM).

Le modèle de concurrence basé sur la mémoire transactionnelle a été initialement proposé par [44] comme une méthode de structuration de processus parallèles en se basant sur les actions atomiques. Le terme action atomique a été ensuite remplacé par le terme transaction pour faire un rapprochement au concept des transactions largement étudié dans le domaine des bases de données. Ainsi, le modèle TM consiste en un ensemble de *threads*, qui s'exécutent dans un espace de mémoire partagé, et qui y accèdent en manipulant des transactions. Une transaction consiste en un bloc d'instructions qui exécutent une série de lecture et écriture dans la mémoire partagée. La propriété d'isolation (atomicité) de la transaction doit être maintenue : les lectures et les écritures, à l'intérieur d'une même transaction, doivent logiquement avoir lieu instantanément, les états intermédiaires ne sont pas visibles aux autres transactions.

Le TM fournit plusieurs avantages par rapport à la programmation parallèle basée sur les mécanismes de bas niveau cité plus haut. Principalement, le TM permet au concepteur d'écrire un programme parallèle plus facilement et les risques d'avoir des erreurs telles que les concurrences critiques («*race condition*») et les

interblocages deviennent minimales. Voici quelques raisons qui montrent l'avantage de TM par rapport à la programmation parallèle classique[44, 24] :

1. La difficulté majeure dans la conception des programmes parallèles est la coordination des accès concurrents à la mémoire partagée par plusieurs *threads*. Concurrence critique (race condition), interblocage et faible performance sont les conséquences dues au renforcement de l'exclusion mutuelle avec peu ou beaucoup de synchronisation. Le TM offre une alternative plus simple à l'exclusion mutuelle en enlevant la charge de gérer la synchronisation explicitement par le programmeur.
2. Grâce au niveau d'abstraction offert par le TM, le concepteur a besoin seulement d'identifier les séquences d'opérations qui doivent être exécutées d'une manière atomique et de les encapsuler à l'intérieur d'une transaction. Ceci permet au concepteur de raisonner sur la validité de la fonctionnalité du code à l'intérieur d'une transaction sans se préoccuper de l'interaction complexe avec les autres transactions du programme qui s'exécutent en parallèle. C'est à l'implémentation du TM de garantir la propriété de l'atomicité des transactions en ayant des performances acceptables.
3. En plus de fournir une meilleure abstraction, les transactions permettent de rendre la synchronisation composable[28]. En effet, en utilisant simplement les verrous comme mécanisme de synchronisation, il n'est pas possible de pouvoir les composer ensuite. Considérons l'exemple de la Figure 3.1 : Une classe qui implémente un compte bancaire.

Cette classe fournit deux opérations (« *Thread-safe*») `deposit` et `withdraw` pour déposer et retirer de l'argent d'un compte bancaire. Supposons que nous voulons composer ces opérations à l'aide des verrous dans une opération transfert, qui transfère de l'argent d'un compte à un autre. Cette opération doit être atomique : L'état intermédiaire où l'argent a été débité, mais pas encore crédité ne doit pas être visible aux autres *threads*. Implémenter l'opération transfert sans risque de concurrence critique (`transfer_`

```

class Account {
    float balance;
    void deposit(float amt) {
        lock (this) {
            balance += amt;
        }
    }
    void withdraw(float amt) {
        lock (this) {
            if (balance < amt)
                throw new OutOfMoneyError();
            balance -= amt;
        }
    }
    void transfer_wrong1(Account other,
        float amt) {
        other.withdraw(amt);
        // race condition:
        // incorrecte somme de balance
        this.deposit(amt);
    }
    void transfer_wrong2(Account other,
        float amt) {
        lock (this) {
            // interblocage lors
            // d'un transfert inverse
            other.withdraw(amt);
            this.deposit(amt);
        }
    }
}

class Account {
    float balance;
    void deposit(float amt) {
        atomic {
            balance += amt;
        }
    }
    void withdraw(float amt) {
        atomic {
            if (balance < amt)
                throw new OutOfMoneyError();
            balance -= amt;
        }
    }
    void transfer(Account other,
        float amt) {
        atomic {
            other.withdraw(amt);
            this.deposit(amt);
        }
    }
}

```

Figure 3.1 Difficulté d'étendre le logiciel avec des verrous[24]

wrong1) ou interblocage (transfert_wrong2) nécessite un changement majeur impliquant une discipline subtile dans l'ordre d'acquisition de verrous pour éviter l'interblocage et permettre à deux opérations indépendantes de s'exécuter concurremment. D'un autre côté, le modèle de concurrence TM permet aux deux opérations d'être composées directement. Les opérations deposit et withdraw s'exécutent dans leurs propres transactions lorsqu'elles sont appelées individuellement. L'opération Transfert s'exécute aussi dans

une transaction. Les transactions imbriquées seront fusionnées dans cette même transaction.

Comme nous venons de voir, les mémoires transactionnelles facilitent considérablement la description d'un programme parallèle. Cependant, toute la subtilité lors de la synchronisation des processus concurrents doit être traitée par un gestionnaire de TM. La prochaine section décrit différents types d'implémentations d'un gestionnaire de TM.

3.1.2 Implémentation de TM

Afin de préserver la propriété d'atomicité d'une transaction, la manière la plus directe afin d'implémenter les mémoires transactionnelles est d'exécuter chaque transaction individuellement jusqu'à ce qu'elle termine avant qu'une autre transaction ne puisse commencer. De cette manière, chaque transaction correspond logiquement à une exécution instantanée, ainsi aucun état intermédiaire ne peut être visible aux autres transactions. Nous allons appeler cette implémentation une exécution séquentielle.

Implémenter TM en tant qu'exécution séquentielle des transactions peut engendrer des performances médiocres, puisque cette implémentation ne peut pas profiter d'une machine multicœur. Par conséquent, il faut considérer une exécution concurrente des transactions en permettant que les opérations à l'intérieur des transactions puissent être entrelacées. Afin qu'une exécution concurrente préserve la propriété d'atomicité de chaque transaction exécutée, elle doit respecter la propriété de **sérialisabilité** décrite ci-dessous. Au chapitre 4 nous allons donner une interprétation formelle de cette notion de sérialisabilité.

sérialisabilité : *Le résultat de l'exécution concurrente des transactions doit correspondre au même résultat que nous pouvons obtenir suite à une exécution séquentielle de ces transactions.*

Plusieurs implémentations de TM ont été étudiées afin de pouvoir exécuter les transactions d'une manière concurrente tout en satisfaisant la propriété de sérialisabilité. Deux approches sont utilisées : soit en considérant un contrôle de concur-

rence optimiste ou pessimiste. Dans l'approche basée sur le contrôle de concurrence optimiste, les transactions s'exécutent sous l'hypothèse que l'exécution est sérialisable. Si le gestionnaire TM détecte, à un moment donné (par exemple, durant le commit d'une transaction), que l'ordonnancement généré n'est pas sérialisable, alors il doit choisir une (ou plusieurs) transaction qui ne satisfait pas la propriété de sérialisabilité puis l'annuler (abort) en annulant les modifications engendrées par son exécution (rollback) puis reprendre son exécution ultérieurement. Contrairement à l'approche optimiste, l'approche pessimiste demande qu'une transaction obtienne un accès exclusif à une zone mémoire, en utilisant des verrous, avant qu'elle ne puisse y accéder. Cette approche doit pouvoir aussi avorter une transaction vu qu'il existe un risque d'interblocage. Dans les deux types de contrôle de concurrence, on retrouve différentes techniques. Un aperçu sur les principales techniques employées a été présenté dans [42, 9] ainsi que les avantages et les inconvénients de chacune. Une vue d'ensemble sur les différentes implémentations de TM proposées dans la littérature est donnée dans [43].

La plupart des systèmes basés sur le TM ajoutent simplement des instructions du genre `atomic{...}` afin que le programmeur puisse regrouper un ensemble d'instructions qui doit être exécuté en une seule transaction. Cependant, les transactions à elles seules fournissent peu d'assistance pour synchroniser plusieurs tâches concurrentes. En effet, prenons l'exemple d'un producteur qui écrit dans une FIFO et un consommateur qui lit à partir de la même FIFO. Nous pouvons isoler la lecture et l'écriture dans deux transactions différentes afin que les accès à la FIFO n'interfèrent pas. Cependant, lorsque la FIFO est pleine, le producteur ne peut pas continuer et doit reprendre la transaction plus tard. Il en va de même pour le consommateur lorsque la FIFO est vide. Sans mécanisme de synchronisation, le producteur doit attendre activement en consultant continuellement l'état de la FIFO jusqu'à ce qu'elle devienne non pleine. Et la même attente active est exécutée par le consommateur. Pour éviter ces attentes actives qui ont un impact considérable sur les performances, il faut permettre que le producteur soit suspendu en attente que la condition sur la FIFO soit satisfaite.

La solution à ce problème de synchronisation est d'affecter à une transaction

une garde qui détermine le moment où celle-ci doit entamer ou continuer son exécution. Le gestionnaire de TM a la responsabilité de réveiller une transaction qui est en attente sur une garde lorsque celle-ci devient vraie. L'idée de synchroniser un programme parallèle en utilisant des gardes n'est pas nouvelle. Par exemple [44] propose l'instruction `await(condition) then {...}` afin de bloquer une transaction jusqu'à ce que la `condition` soit vraie. Les auteurs dans [30] proposent le concept de région critique conditionnée («*Conditional Critical Region*») qui est similaire à la notion d'action atomique gardée où une région est gardée par une condition booléenne. Cependant, implémenter le modèle TM[28, 27, 19] en ajoutant le concept de synchronisation basée sur des gardes rend le problème encore difficile, puisqu'il ne suffit pas de garantir la sérialisabilité de l'exécution, mais aussi de gérer toute la synchronisation nécessaire entre les transactions concurrentes et de savoir quand une transaction doit reprendre son exécution suite à une attente sur une garde. Les travaux de [27] ont présenté une implémentation en Java en ajoutant au langage une instruction `atomic`. Par exemple, une méthode qui lit une valeur d'une FIFO peut être programmée comme suit :

```
Item get() {  
    atomic (n_items > 0) { /* remove item */  
    }  
}
```

Il est aussi possible que deux transactions soient composées. Par exemple, le code suivant retire deux éléments consécutifs d'une FIFO.

```
atomic { item1 = get() ; item2 = get(); }
```

Le problème avec la composition des transactions est qu'une transaction peut s'exécuter longtemps avant de s'apercevoir qu'elle ne peut plus avancer. Dans ce cas le gestionnaire doit l'avorter et attendre que la condition soit satisfaite. Selon [27], indépendamment des variables qui déterminent la condition dans `atomic(condition)`, lorsqu'une transaction doit attendre sur une condition, le gestionnaire annule la transaction et la suspend, en enregistrant toutes les variables qui ont été accédées en lecture depuis son début d'exécution jusqu'à sa suspension. Si l'une de ces variables a été modifiée par une autre transaction, le gestionnaire relance la transaction en attente. Ceci dégrade encore les performances si une

transaction se réveille et s’aperçoit que la condition pour pouvoir continuer n’est plus satisfaite, ainsi elle doit être annulée et suspendue encore une fois. Une autre implémentation de TM avec Haskell a été proposée dans [28]. Deux instructions `retry` et `orElse` sont fournies pour pouvoir attendre sur une condition et choisir parmi deux transactions à exécuter. L’instruction `retry` suspend la transaction en cours et l’annule. Elle sera relancée lorsqu’une variable lue précédemment a été modifiée. C’est la même technique utilisée dans [27], sauf que maintenant le programmeur fait une demande de suspension explicitement à l’aide de l’instruction `retry`. L’instruction `orElse` permet de composer deux transactions en choisissant une deuxième transaction si la première est suspendue.

Nous allons utiliser les mémoires transactionnelles, en tant que mécanisme de contrôle de concurrence, dans le but de modéliser les systèmes sur puces à plusieurs niveaux d’abstraction. Ainsi, un système sur puce va être décrit par un modèle transactionnel constitué de plusieurs transactions qui communiquent et se synchronisent entre elles à travers des variables partagées. Les détails sur l’implémentation et la structure de ce modèle sont décrits dans la section 5.1 du chapitre 5. La prochaine section décrit comment un modèle RTL est généré à partir d’un modèle transactionnel.

3.2 Synthèse de matériel

Dans cette section, nous allons montrer comment un modèle transactionnel est transformé en une description RTL synthétisable en se basant sur l’approche de synthèse décrite dans [32, 31]. Ce travail était à l’origine de l’outil de synthèse Bluespec [12].

Considérons un modèle transactionnel spécifiée selon le langage des commandes gardées de Dijkstra [9] :

$$MT \equiv \llbracket \text{var } s; S_0; \mathbf{do} \ T_1 \parallel T_2 \parallel \dots \parallel T_m \ \mathbf{od} \rrbracket$$

Le N-uplet s spécifie les variables partagées qui forment l’état du modèle transactionnel MT . S_0 est l’instruction d’initialisation de ces variables et T_1, T_2, \dots, T_m

sont les transactions (commandes gardées) de MT . Chaque transaction T_i est de la forme suivante :

$$T_i \equiv \eta_i(s) \Rightarrow s' := \delta_i(s)$$

Où $\eta_i(s)$ est la garde de la transaction T_i . La garde $\eta_i(s)$ est une expression booléenne évaluée sur les valeurs des variables d'états s . $\delta_i(s)$ est la fonction qui calcule l'état prochain s' du système à partir de l'état courant s .

La sémantique d'exécution du modèle transactionnel est de sélectionner d'une manière non déterministe une transaction dont la garde est vraie puis l'exécuter. L'exécution continue tant qu'il existe une garde qui est évaluée à vraie :

Tant que (il existe une garde $\eta_i(s)$ vraie)

1. Choisir une transaction T_i telle que $\eta_i(s)$ est vraie.
2. exécuter $s' := \delta_i(s)$

3.2.1 Implémentation de référence

Il existe une transformation directe d'un modèle transactionnel vers une implémentation matérielle synchrone que nous appelons implémentation de référence. Cette implémentation de référence considère les deux conditions suivantes :

1. Chaque transaction est exécutée au sein d'un même cycle d'horloge.
2. Au maximum une transaction est exécutée durant un cycle.

La génération de la description RTL de cette implémentation de référence est faite selon les étapes suivantes :

1. **Extraction de $\eta_i(s)$ et $\delta_i(s)$** : Chaque garde $\eta_i(s)$ et fonction de mise à jour $\delta_i(s)$ est implémentée en tant que logique combinatoire.
2. **Ordonnanceur** : Un ordonnanceur est ajouté de telle façon qu'à chaque cycle il choisit une transaction à exécuter parmi plusieurs transactions dont les signaux $\delta_i(s)$ sont actifs. Pour chaque transaction, un signal ϕ_i est alloué. Ainsi, l'ordonnanceur active un seul signal ϕ_i reflétant la transaction A_i

qui a été choisie. Un encodeur prioritaire est un ordonnanceur valide pour l'implémentation de référence.

3. **Registres** : À chaque variable d'état s_k correspond un registre. Ce registre admet deux entrées **LE** et **D**. Lorsque le signal **LE** est actif, le registre prend sa nouvelle valeur à partir de l'entrée de donnée **D**. Autrement il garde son ancienne valeur. Plusieurs transactions ($m \leq n$) peuvent écrire sur le registre. Le registre est mis à jour si une de ces transactions est active. Ainsi, le signal **LE** prend comme entrée le ou-logique de tous les signaux $\phi_i, 1 < i < m$, des m transactions correspondantes. La nouvelle valeur du registre est sélectionnée à l'aide d'un multiplexeur contrôlé par les signaux ϕ_i , et ceci à partir des m valeurs représentées par les signaux $\delta_i(s)_{s_k}$. La Figure 3.2 montre le circuit RTL d'un registre représentant la variable s_k où deux transactions T_1 et T_2 écrivent dessus.

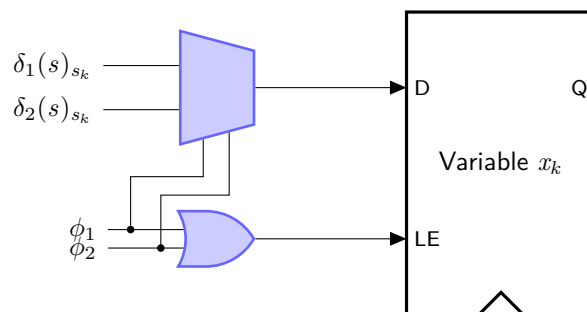


Figure 3.2 Circuit RTL d'un registre correspond à une variable accédée par deux transactions

La Figure 3.3 montre le circuit généré à partir d'un modèle transactionnel suite aux étapes décrites plus haut.

l'Exemple 3.1 montre une description d'un modèle transactionnel contenant deux variables d'états x et y et qui calcule le plus grand diviseur commun en utilisant l'algorithme d'Euclid.

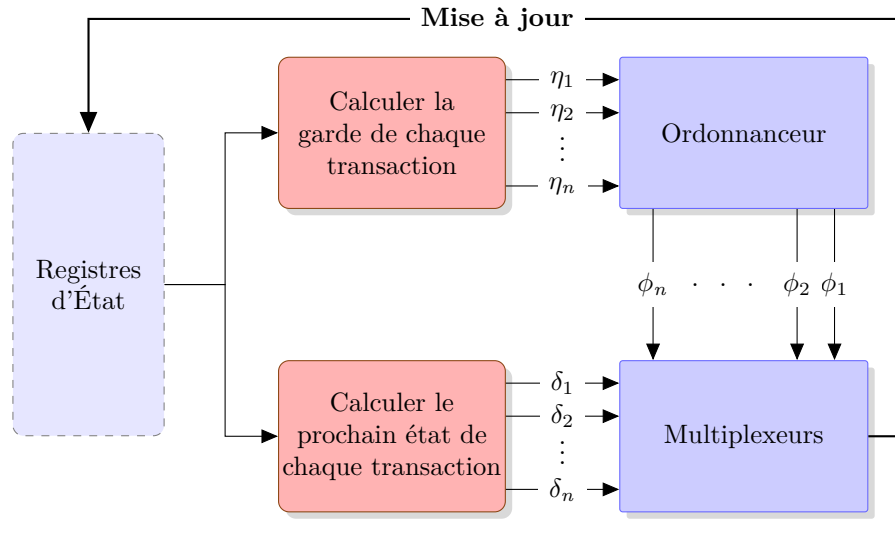


Figure 3.3 Circuit RTL généré à partir d'un modèle transactionnel

Exemple 3.1 *Modèle transactionnel GCD*

Soit le modèle transactionnel $\mathbf{gcd}(p, q)$ suivant qui calcule le plus grand commun diviseur de deux entiers p et q :

$$\mathbf{gcd}(p, q) \equiv \llbracket \text{var } (x, y); (x, y) = (p, q); \mathbf{do } T_1 \parallel T_2 \mathbf{od} \rrbracket$$

où

$$T_1 \equiv (x < y) \quad \Rightarrow \quad (x, y)' := (y, x)$$

$$T_2 \equiv (x \geq y \wedge y \neq 0) \quad \Rightarrow \quad (x, y)' := (x - y, y)$$

La Figure 3.4 illustre le circuit synthétisé à partir de la description $\mathbf{gcd}(p, q)$ du modèle transactionnel.

Il est important de noter que les deux transactions T_1 et T_2 sont mutuellement exclusives. C'est-à-dire que les signaux η_1 et η_2 ne peuvent pas être actifs dans le même cycle d'horloge. Ainsi, il est possible d'éliminer l'ordonnanceur et

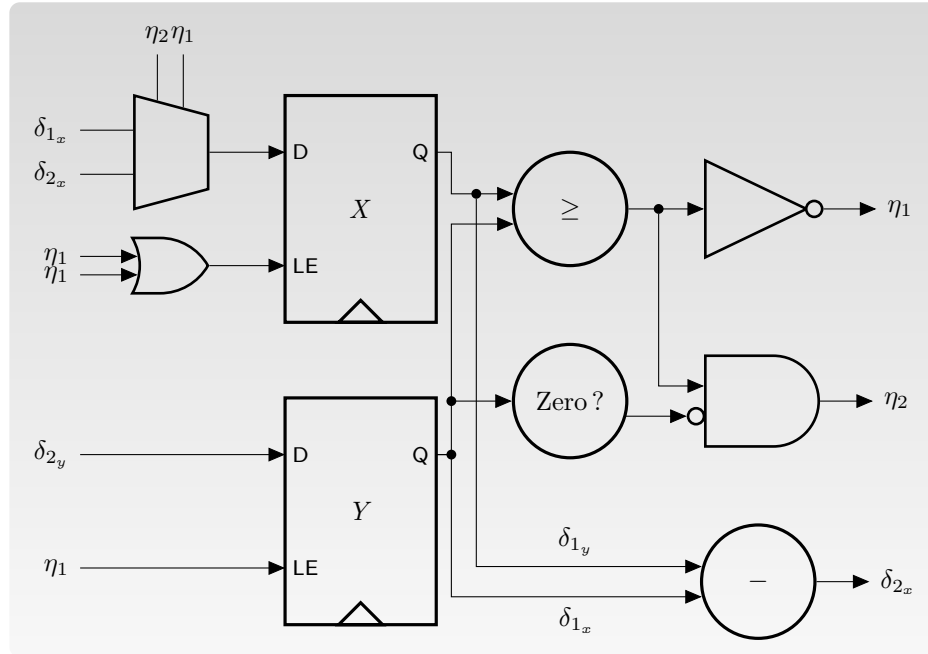


Figure 3.4 Circuit généré à partir du modèle transactionnel $\text{gcd}(p, q)$

utiliser directement les signaux η_1 et η_2 au lieu de ϕ_1 et ϕ_2 . Par conséquent, une analyse des transactions pour déterminer si elles sont mutuellement exclusives s'avère nécessaire afin d'éliminer les cas d'ordonnement qui ne peuvent pas être présents logiquement et ainsi minimiser la quantité de matériel requis et la durée du cycle d'horloge. Aussi il est possible de faire des optimisations de la logique combinatoire. Par exemple, les calculs $x < y$ et $x \geq y$ peuvent partager le même circuit correspondant à $x \geq y$



Exactitude de l'implémentation de référence

L'implémentation de référence présentée plus haut n'est pas complètement équivalente au modèle d'exécution d'un modèle transactionnel. En effet, dans l'implémentation de référence, à moins que l'ordonnanceur choisisse arbitrairement une transaction à chaque cycle, l'implémentation est toujours déterministe.

En d'autres termes, l'implémentation exécute un seul comportement parmi plusieurs admis par le modèle transactionnel. Un circuit synthétisé qui implémente un modèle transactionnel doit satisfaire les deux conditions suivantes :

1. La séquence des transitions d'un état à un autre doit correspondre à une des séquences permise par le modèle transactionnel. Ceci est valide pour l'implémentation de référence puisqu'à chaque cycle une seule transaction est exécutée. Ainsi, la propriété d'atomicité est préservée. La séquence des valeurs d'états à chaque top d'horloge correspond à une séquence d'états du modèle transactionnel.
2. L'implémentation doit garantir la vivacité du système. Autrement une implémentation qui ne change pas l'état du système serait une implémentation correcte. En général la vivacité des transitions est maintenue pour tout ordonnanceur qui satisfait la propriété suivante :

$$\eta_1 \vee \eta_2 \vee \dots \vee \eta_n \Rightarrow \phi_1 \vee \phi_2 \vee \dots \vee \phi_n$$

L'encodeur par priorité maintient la vivacité puisqu'il active un signal ϕ à chaque fois qu'au moins un des signaux η est actif.

3.2.2 Optimisations

Bien que la sémantique d'exécution d'un modèle transactionnel ainsi que l'implémentation de référence décrive une exécution séquentielle des transactions, une implémentation matérielle peut exécuter plusieurs transactions parallèlement dans un même cycle d'horloge. Cependant dans une implémentation avec multiples transactions par cycle, la transition d'états prise dans chaque cycle doit correspondre à une certaine exécution séquentielle du modèle transactionnel.

Théoriquement, n'importe quelles deux transactions peuvent être composées pour qu'elles s'exécutent parallèlement pendant le même cycle. Il suffit de permettre à une transaction de lire les changements effectués par l'autre transaction. Cependant, synthétiser un circuit en permettant que les transactions communiquent entre elles dans un même cycle, et avoir ainsi une cascade de logique

combinatoire, a pour résultat d'augmenter considérablement la taille de circuit et la durée du cycle d'horloge. En effet, supposons un modèle transactionnel décrivant un pipeline où chaque transaction spécifie une fonction dans une des phases du pipeline. Synthétiser ce système, en permettant le transfert des données entre les transactions, génère un circuit non-pipeline ayant une seule phase qui est la composition séquentielle de toutes les phases précédentes. Néanmoins, il existe des situations où il est préférable d'exécuter une transaction qui dépend des résultats d'une autre transaction dans le même cycle. Une des situations est, par exemple, le «*bypassing*» des valeurs que nous retrouvons dans la plupart des processeurs pipelines.

L'approche de synthèse, proposée par [31], permet de générer un ordonnanceur capable d'exécuter plusieurs transactions dans le même cycle sans permettre la communication intra-cycle entre les transactions. Cette approche est basée sur l'analyse statique des transactions afin de déterminer si deux transactions sont non conflictuelles (CF) ou si elles peuvent être composées séquentiellement (SR).

Deux transactions A et B sont non conflictuelles, notées $A <_{CF} B$, si elles ne lisent pas et n'écrivent pas la même variable. Soit :

$$\begin{aligned} \text{Read}(A) \cap \text{Write}(B) &= \{\} \text{ et} \\ \text{Read}(B) \cap \text{Write}(A) &= \{\} \text{ et} \\ \text{Write}(B) \cap \text{Write}(A) &= \{\} \end{aligned}$$

Dans ce cas les deux transactions peuvent être exécutées en même cycle et leur exécution correspond à une exécution de A suivie de B ou inversement. Par exemple, étant donné les transactions A et B suivante :

$$\begin{aligned} A &\equiv (\text{true}) \Rightarrow x' := x + 1 \\ B &\equiv (\text{true}) \Rightarrow y' := y + 2 \end{aligned}$$

L'exécution concurrente de A et B dans le même cycle correspond au comportement de la séquence composée C suivante. C va lire l'état $s = (x, y)$ au début

du cycle puis met à jour l'état $s' = (x + 1, y + 2)$ à la fin du cycle. L'état obtenu est le même que celui obtenu suite à une exécution séquentielle de A suivie de B ou inversement, i.e, $\delta_A(\delta_B(s)) = \delta_B(\delta_A(s))$. Par conséquent, puisque l'exécution concurrente de A et B correspond à une exécution séquentielle de A suivie de B (ou inversement), l'ordonnanceur généré peut permettre l'activation de ces deux transactions au même cycle.

Une transaction A est séquentiellement composable avec une transaction B , noté $A <_{SC} B$, si B ne lit aucune variable que A écrit. Soit :

$$\text{Read}(B) \cap \text{Write}(A) = \{\}$$

La procédure de synthèse va ignorer les mises à jour faites par A sur les registres qui sont aussi mis à jour par B au cours du même cycle. Par exemple, étant donné les transactions A et B suivante :

$$\begin{aligned} A &\equiv (\text{true}) \Rightarrow x' := y + 1 \\ B &\equiv (\text{true}) \Rightarrow y' := y + 2 \end{aligned}$$

L'exécution concurrente de A et B dans le même cycle correspond au comportement de la séquence composée C suivant. C va lire l'état $s = (x, y)$ au début du cycle puis met à jour l'état $s' = (y + 1, y + 2)$ à la fin du cycle. L'état obtenu serait le même suite à une exécution séquentielle de A suivie de B , ainsi $A <_{SC} B$. Cependant, contrairement à l'exemple précédent la relation $B <_{SC} A$ n'est pas valide.

L'idée de [31] est de partitionner l'ensemble des transactions en plusieurs groupes, appelés groupes d'arbitrage. Deux transactions qui appartiennent à deux groupes différents sont séquentiellement composables et ainsi elles peuvent être exécutées dans le même cycle. La procédure de synthèse commence par générer un graphe où chaque nœud correspond à une transaction (voir Figure 3.5(a)). Un arc existe entre un nœud A et un nœud B si $(A <_{SC} B)$ et $\neg(A <_{CF} B)$. La deuxième étape est de rendre le graphe acyclique (Figure 3.5(b)). Un cycle dans

ce graphe correspond à la situation où les transactions appartenant à ce cycle ne peuvent pas s'exécuter dans le même cycle d'horloge suite à une dépendance de données circulaire entre elles. La troisième étape est de générer les groupes d'arbitrage. Un arc existe entre deux nœuds A et B s'il n'existe aucun arc qui les relie dans le graphe obtenu précédemment et $\neg(A <_{CF} B)$. Les composants connectés du graphe de conflit représente les différents groupes d'arbitrage. Chaque groupe d'arbitrage va correspondre à un à ordonnanceur local qui gère seulement les transactions qui appartiennent à ce groupe. Un encodeur de priorité est un ordonnanceur valide. Cependant certaines optimisations peuvent être faites. Par exemple, dans le groupe d'arbitrage 1 de la Figure 3.5(c), si T_4 n'est pas activée alors l'ordonnanceur peut activer les transactions T_1 , T_2 et T_5 au même cycle. Une table de recherche de taille $n * 2^n$ indexée par les signaux η_i peut être implémentée afin de déterminer les valeurs des signaux ϕ_i .

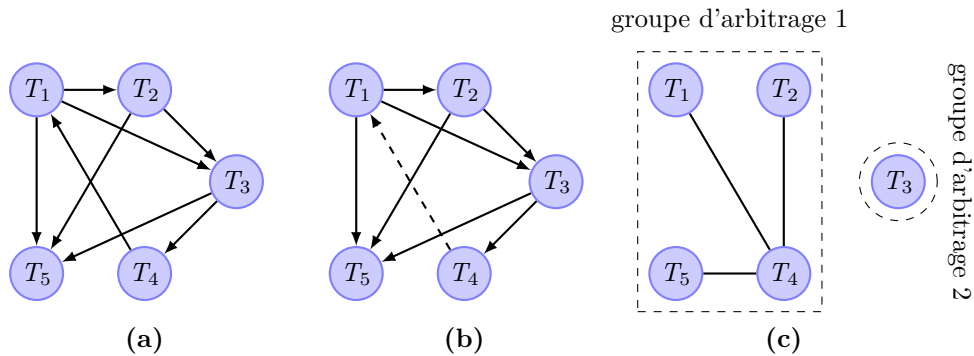


Figure 3.5 Analyse des transactions. (a) graphe $<_{SC}$ orienté, (b) graphe $<_{SC}$ acyclique, (c) groupe d'arbitrages

3.2.3 Bluespec

Bluespec est un langage basé sur les transactions (règles de réécriture de termes) permettant de synthétiser automatiquement un modèle de transactions en une implémentation RTL. Cependant, plusieurs phases dans le cycle de conception restent à explorer.

- Une transaction est censée s'exécuter pendant un cycle. Par conséquent,

avant qu'un système ne puisse être synthétisable, il faut que le modèle soit précis au niveau cycle (Cycle Accurate, CA) et que le détail de la microarchitecture soit bien établi. Un concepteur peut entamer la conception de son système directement en utilisant un modèle au niveau cycle, ou en commençant par un modèle de haut niveau puis le raffiner à un modèle au niveau cycle. Dans le premier cas, Bluespec n'offre pas encore les mécanismes nécessaires pour vérifier que le modèle est conforme à certaines propriétés. Il est encore possible d'utiliser des langages d'assertions tels que SVA ou PSL une fois le modèle RTL généré. Cependant, plusieurs signaux de contrôle sont générés automatiquement et il est difficile de faire le lien entre le modèle Bluespec original et le modèle RTL. Dans le deuxième cas, le raffinement d'un modèle de plus haut niveau à un modèle synthétisable est géré manuellement par le concepteur et l'absence d'outils de vérification ne permet pas de valider que le modèle de bas niveau satisfait la spécification initiale de haut niveau.

- Malgré que le point d'entrée pour la synthèse d'un système Bluespec soit un modèle cycle-accurate, il reste encore une description de haut niveau comparée à un modèle RTL. En effet, un modèle transactionnel est un modèle non déterministe puisqu'il peut y avoir plusieurs transactions actives pendant un cycle et il faut choisir une d'une façon non déterministe. Cependant, une implémentation RTL est déterministe et correspond à un ordonnancement déterministe des transactions à chaque cycle. Laisser le compilateur générer un ordonnanceur automatiquement sans intervention du concepteur va produire un modèle fonctionnellement correct, cependant certaines propriétés non fonctionnelles telles que la latence ou la largeur de bande peuvent être insatisfaisantes. La nouvelle version de Bluespec fournit au concepteur des annotations d'ordonnancement déclaratives afin d'avoir plus de contrôle sur l'ordonnancement des transactions et ainsi pouvoir contrôler l'aspect non fonctionnel du modèle RTL à générer. Toutefois, une fois que le concepteur a décidé comment ordonnancer le modèle en une implémentation détermi-

niste, il faut avoir les outils nécessaires afin de valider que son choix satisfait certaines propriétés temporelles.

À la fin de cette thèse (chapitre 7), nous proposons un flot de conception complet de systèmes sur puces qui est basé sur le concept de transactions. Nous avons montré comment Bluespec peut être intégré dans ce flot de conception. L'avantage de cette méthodologie est qu'elle se repose sur des raffinements incrémentaux rendus possibles grâce à la sémantique formelle des modèles transactionnels qui fait l'objet de la prochaine section.

3.3 Raisonnement formel

Il existe plusieurs formalismes afin de raisonner sur les modèles transactionnels. Par exemple, UNITY[50] et TLA (Temporal Logic of Actions)[40] se basent sur la logique temporelle. Les systèmes d'actions dans [6] raisonnent avec la sémantique de «*weakest precondition*» [21] afin de définir un calcul de raffinement des systèmes parallèles [8]. Tous ces formalismes partagent le même concept de transaction avec quelques différences au niveau des opérations utilisées au sein des transaction et des propriétés qui peuvent être exprimées. Par la suite nous introduisons la logique TLA suivie de la notion de raffinement d'une spécification à une implémentation.

Nous avons choisi TLA parce que celle-ci vient avec les outils nécessaires [39] pour faciliter la vérification tels qu'un ModelChecker pour la vérification automatique et une formulation de TLA dans un système de preuves pour le raisonnement déductif[48].

3.3.1 Aperçu de TLA

La logique temporelle des actions (TLA)[40] est une logique permettant de raisonner sur les systèmes concurrents et réactifs. Cette logique a été étendue par des constructions syntaxiques permettant la structuration de la spécification et l'introduction des types de données formant ainsi le langage formel TLA+. Afin d'introduire TLA+ nous présentons un système qui affiche les heures d'une horloge[39]. Prenons la variable `hr` comme étant l'afficheur de l'heure de l'horloge.

Un comportement typique de l'horloge correspond à la séquence d'états suivante :

$$[hr = 11] \mapsto [hr = 12] \mapsto [hr = 1] \mapsto [hr = 2] \mapsto \dots$$

Deux états successive tels que $[hr = 11] \mapsto [hr = 12]$ est appelé transition. Afin de spécifier l'horloge, il faut décrire tous ses comportements possibles. Pour cela, il faut définir un prédicat initial spécifiant les valeurs initiales possibles de la variable hr puis une relation entre ces deux états successifs spécifiant les changements possibles de hr au cours d'une transition. Initialement, hr peut avoir n'importe quelle valeur entre 1 et 12. Ceci est exprimé par le prédicat $HCini$ suivant :

$$HCini \triangleq hr \in (1 .. 12)$$

La relation entre deux états successifs est définie par une formule qui relie la valeur de hr entre l'état courant et le prochain état. Soit hr la valeur de hr dans l'état courant et hr' la valeur de hr dans le prochain état. Nous voulons que la valeur de la variable hr soit incrémentée en cas où hr est différente de 12, autrement elle est initialisée à 1. Ceci est exprimé par la formule $HCnext$ suivante :

$$HCnext \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$$

La formule $HCnext$ est une formule mathématique ordinaire, excepté qu'elle contient des variables primées et non primées. Telle formule est appelée une action. Une action est vraie ou faux lors d'une transition. Donc une action est évaluée entre deux états successifs d'un comportement.

Afin de compléter la spécification de l'horloge, il faut définir une seule formule qui décrit tous les comportements possibles de l'horloge. Ces comportements doivent satisfaire les conditions suivantes :

1. L'état initial doit satisfaire le prédicat $HCini$.
2. Chaque de transition doit satisfaire l'action $HCnext$. Ceci est exprimé par l'opérateur de la logique temporelle \square . La formule temporelle $\square F$ affirme que la formule F est toujours vraie. En particulier $\square HCnext$ affirme que l'action $HCnext$ est vraie dans chaque transition du système.

Ces deux conditions sont satisfaites en définissant la formule HC comme conjonction du prédicat $HCini$ et la formule Ces deux conditions sont satisfaites en définissant la formule HC comme conjonction du prédicat $HCini$ et la formule $\Box HCnext$:

$$HC \triangleq HCini \wedge \Box HCnext$$

Si nous considérons l'horloge en isolation sans la relier à un autre système, alors la spécification précédente de l'horloge est une spécification satisfaisante. Cependant, supposons que l'horloge fait partie d'un système plus large qui affiche l'heure aussi bien que la température. L'état de ce système est représenté par une variable hr affichant l'heure et une variable tmp affichant la température. Considérons le comportement possible du nouveau système :

$$\left[\begin{array}{l} hr = 11 \\ tmp = 23.5 \end{array} \right] \mapsto \left[\begin{array}{l} hr = 12 \\ tmp = 23.5 \end{array} \right] \mapsto \left[\begin{array}{l} hr = 12 \\ tmp = 23.4 \end{array} \right] \mapsto \dots$$

Dans la deuxième et la troisième transition, la variable tmp change alors que la variable hr garde sa valeur égale à 12. Ces transitions ne satisfont pas l'action $HCnext$ qui affirme que la variable hr doit être incrémentée. Ainsi, la spécification HC ne permet pas de décrire une horloge dans ce nouveau système. Pour remédier à ce problème, chaque formule qui décrit l'horloge doit permettre des transitions où la valeur de h ne change pas. Ces transitions sont appelées «*stuttering*» transitions. Ainsi, la spécification HC doit être changée par la formule suivante :

$$HC \triangleq HCini \wedge \Box [HCnext \vee (hr' = hr)]$$

Où d'une façon plus compacte :

$$HC \triangleq HCini \wedge \Box [HCnext]_{hr}$$

La spécification complète de l'horloge est définie dans le module TLA+ HourClock suivant :

MODULE *HourClock*

EXTENDS *Naturals*

VARIABLE *hr*

$$\begin{aligned}
HCini &\triangleq hr \in (1 \dots 12) \\
HCnext &\triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1 \\
HC &\triangleq HCini \wedge \square[HCnext]_{hr}
\end{aligned}$$

D'une façon générale un système est décrit par la formule TLA de la forme suivante :

$$Init \wedge \square[Next]_{vars} \wedge Fairness$$

1. **Init** : C'est un prédicat qui spécifie les états initiaux du système de transition.
2. **Next** : C'est la disjonction de plusieurs actions A_i qui spécifient les transitions possibles d'un état à un autre. Elle a la forme générale : $A_1 \vee A_2 \vee \dots \vee A_m$
3. **Fairness** : Elle décrit des propriétés de vivacité que le système doit respecter. Par exemple dans la spécification HC de l'horloge il est permis que la valeur de hr reste inchangée indéfiniment. Une propriété de vivacité affirme que le système est toujours actif.
4. **vars** : Afin de spécifier des transitions qui ne changent pas les valeurs de vars.

Il est important de noter la correspondance entre une formule TLA et un modèle transactionnel spécifiée selon le langage des commandes gardées de Dijkstra (voir section 3.2) de la forme suivante :

$$\begin{aligned}
MT &\equiv \llbracket \text{var } s; S_0; \\
&\mathbf{do} \\
&\quad \eta_1(s) \Rightarrow s' := \delta_1(s) \\
&\quad \parallel \quad \eta_2(s) \Rightarrow s' := \delta_2(s) \\
&\quad \parallel \quad \dots \\
&\quad \parallel \quad \eta_n(s) \Rightarrow s' := \delta_n(s) \\
&\mathbf{od} \rrbracket
\end{aligned}$$

- Le N-uplet s correspond aux N-uplet $vars$.
- L'instruction d'initialisation S_0 correspond au prédicat $Init$.
- L'opérateur temporel \square correspond à l'instruction **do**.
- Pour chaque transaction $\eta_i(s) \Rightarrow s' := \delta_i(s)$ du modèle transactionnel correspond l'action TLA : $\eta_i(s) \wedge s' = \delta_i(s)$.

TLA ajoute des propriétés de vivacité qui ne sont pas exprimées dans le modèle transactionnel. Par exemple, supposons qu'un modèle transactionnel contient deux transactions A et B qui sont toujours actives. Un comportement possible de ce système est d'exécuter indéfiniment la transaction A . Les propriétés de vivacité permettent d'empêcher ce genre de comportements afin d'éviter des «*livelock*».

3.3.2 Raffinement

La conception au niveau système, particulièrement TLM(voir chapitre 2), permet de modéliser un système sur puce en utilisant plusieurs niveaux d'abstraction. Nous commençons par un modèle à un haut niveau d'abstraction algorithmique qui décrit simplement la fonctionnalité du système, puis nous descendons dans les niveaux en ajoutant au fur et à mesure des détails d'architecture et de miroarchitecture jusqu'à obtenir un modèle RTL qui décrit l'implémentation du système à synthétiser. Cependant, cette approche perd de son efficacité si le passage d'un niveau à un autre se fait sans pouvoir vérifier que le raffinement fait est correct.

En se basant sur une méthodologie utilisant des modèles transactionnels à tous les niveaux d'abstraction du cycle de développement, alors le passage d'un modèle transactionnel à un autre peut être établi selon des règles de raffinement formelles [2, 7, 5] qui garantissent que le modèle généré préserve les fonctionnalités du modèle initial. Ces règles de raffinement peuvent être classées informellement selon 3 types :

- Raffinement de données : permet de transformer un modèle dans lequel une structure de donnée abstraite est remplacée par une plus concrète.

- Raffinement de l'atomicité : permet de transformer un modèle avec des atomicités à grande granularité vers un autre modèle avec des atomicités à petite granularité. C'est-à-dire qu'une transaction nécessitant la synchronisation et le blocage de plusieurs processus pendant une durée prolongée, limitant ainsi le parallélisme, est remplacée par des transactions de plus petite taille. Selon l'architecture cible sur laquelle le système sera exécuté, le concepteur procède par des étapes de raffinement graduelles jusqu'à arriver à un modèle transactionnel qui reflète directement le système à implémenter sur cette architecture.
- Raffinement par superposition : C'est une transformation d'un modèle afin d'ajouter de nouvelles variables ou modifier d'anciennes variables. Ceci est utile si nous voulons ajouter plus de détails à une spécification pour la rendre moins abstraite. C'est utile aussi si nous voulons modifier une variable, partagée par plusieurs transactions, en plusieurs variables afin d'avoir plus d'indépendance entre les transactions et ainsi augmenter le parallélisme.

L'Annexe I présente un exemple de raffinement d'une spécification abstraite d'une FIFO vers une implémentation dans un langage procédural. À chaque étape de raffinement, nous nous sommes basé sur TLC, le « Model Checker » de TLA+, afin de prouver que notre raffinement est correct.

Dans ce chapitre, nous avons passé en revue plusieurs travaux faits autour des transactions en montrant comment ils peuvent être intégrés dans le cycle de développement des systèmes sur puce sur différents aspects tels que la modélisation, la simulation, la synthèse et la vérification formelle. Dans les prochains chapitres, nous allons montrer comment appliquer la notion de transaction pour développer un environnement de modélisation et simulation parallèle qui profite d'une machine multicœur pour améliorer le temps de simulation. Avant de présenter cet environnement, nous allons tout d'abord exposer dans le chapitre qui suit l'étude théorique sur la base de laquelle nous pouvons valider que notre simulateur parallèle est correct.

Validité d'une exécution concurrente

Dans ce chapitre, nous introduisons les différentes notions de validité permettant de déterminer quand une exécution concurrente de plusieurs transactions est considérée comme correcte. La majorité de ces notions présentées dans ce chapitre sont prises du livre “Transactional Information Systems”[62] qui étudie les transactions dans les systèmes d'information transactionnels en général et les systèmes des bases de données en particulier. Nous avons adapté ces notions à notre environnement de conception des systèmes sur puce afin de constituer une base théorique pour le développement d'ordonnanceurs parallèles.

Nous introduisons d'abord le modèle transactionnel sur lequel nous allons nous baser ainsi qu'une vue conceptuelle de l'environnement d'exécution dans lequel ces transactions sont simulées. Ensuite, nous définissons les différentes notions de sérialisabilité afin de valider qu'une exécution concurrente soit correcte. Finalement, en nous basant sur ces notions de validité, nous présentons les stratégies d'ordonnement que nous avons implémentées (voir chapitres 5 et 6) pour permettre la simulation parallèle d'un modèle transactionnel.

4.1 Environnement d'exécution

Nous supposons que notre modèle transactionnel contient un ensemble fini D , $D = \{x, y, z, \dots\}$, d'entités indivisibles et disjointes stockées en mémoire, appelées variables, sur lesquelles nous pouvons exécuter des opérations d'écriture et de lecture d'une manière indivisible (atomique). Chaque opération de lecture ou d'écriture s'exécute au sein d'un bloc d'opérations décrivant une unité indivisible appelée transaction. Chaque transaction possède une garde constituée d'une condition booléenne qui spécifie quand la transaction est active et prête

pour être exécutée. Nous supposons que notre modèle contient un ensemble fini, $\{T_1, T_2, \dots, T_n\}$, de transactions.

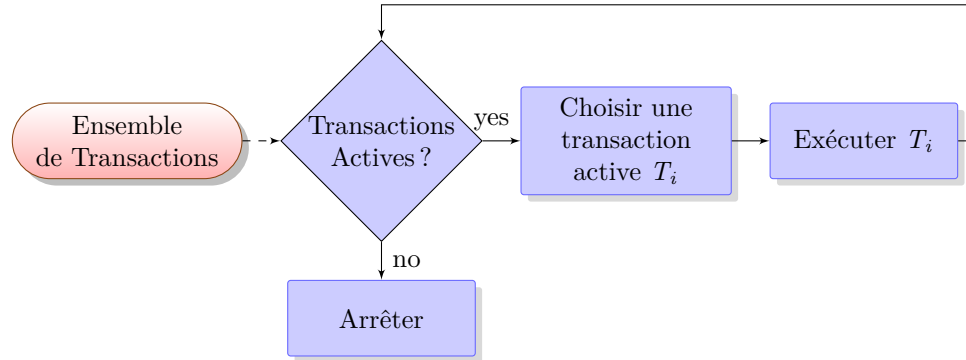


Figure 4.1 Sémantique d'Exécution d'un Modèle Transactionnel

La sémantique d'exécution d'un modèle transaction est illustrée dans la Figure 4.1. Cette sémantique est définie par un programme séquentiel qui réitère tant qu'il existe au moins une transaction active. Durant chaque itération, le programme choisit d'une façon non déterministe une transaction parmi les transactions actives puis l'exécute. Lorsqu'il n'existe plus aucune transaction active, le programme termine.

Remarquer que le programme illustré dans la Figure 4.1 est utilisé pour définir la sémantique d'exécution du modèle transactionnel et non pour imposer une exécution séquentielle lors de la simulation ou l'implémentation de ce modèle. En fait, c'est cette sémantique qui rend le concept des transactions attrayant. En effet, ceci permet au concepteur de raisonner et de valider le comportement d'une transaction d'une manière isolée par rapport aux autres transactions en considérant une transaction comme une simple instruction atomique et indivisible.

Cependant, pour des raisons de performances nous avons intérêt à pouvoir exécuter un modèle transactionnel d'une manière concurrente et ainsi pouvoir profiter d'une machine multicœur. Pour cela, nous supposons que notre environnement d'exécution (voir Figure 4.2) est composé d'un ensemble de *threads* qui exécutent des transactions d'une manière concurrente. Au cœur de cet environnement existe un gestionnaire de transactions qui a la responsabilité de contrôler

la concurrence entre les différents *threads* et d'intercepter les demandes issues de ces derniers et d'y répondre. Chaque *thread* peut exécuter 4 types de demande au gestionnaire de transactions :

1. Demande d'amorçage d'une transaction, notée b . En ce moment, le gestionnaire de transaction assigne à cette nouvelle transaction une valeur unique i . Ensuite, chaque fois qu'un processus fait une demande au gestionnaire de transactions il précise aussi l'indice de transaction à laquelle cette demande appartient.
2. Demande de lecture d'une entité x , notée $r_i(x)$.
3. Demande d'écriture d'une entité x , notée $w_i(x)$.
4. Demande d'abandon(abort), notée a_i .
5. Demande de validation(commit) de la transaction t_i , notée c_i .

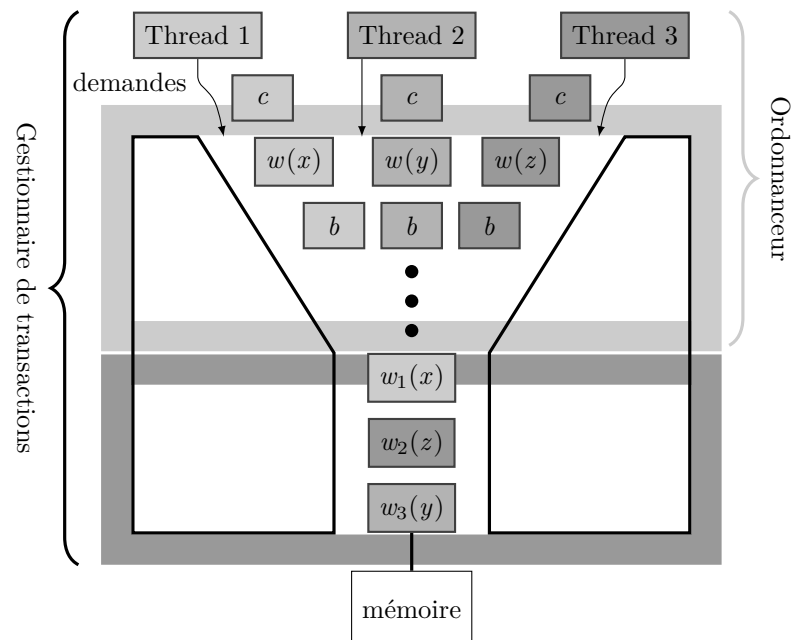


Figure 4.2 Environnement d'exécution d'un modèle de transactions.

Au sein du gestionnaire de transaction est implémenté un ordonnanceur qui a la responsabilité de contrôler la concurrence entre les transactions. En conséquence, pour des raisons de validité, il décide des actions à appliquer en réponse aux demandes interceptées par le gestionnaire de transactions de la manière suivante :

1. **Accepter** la demande. Chaque demande d'amorçage, de lecture (r), d'écriture (w), d'abandon (a) ou de validation (c) est traduite par son action correspondante. Dans ce cas, les actions appliquées aux données peuvent être b , r , w , a , ou c .
2. **Rejeter** la demande. L'ordonnanceur peut rejeter une action de lecture, écriture ou validation pour une raison de validité et la traduit par une action d'abandon. Une demande d'abandon est toujours immédiatement traduite par une action d'abandon.
3. **Bloquer** la demande. La demande (b , r , w ou c) est ni acceptée ni rejetée, mais considérée comme étant non exécutable à cet instant. La décision est reportée pour plus tard. Par exemple, un ordonnanceur peut décider que deux transactions A et B ne peuvent pas procéder en parallèle pour des raisons de validité. Ainsi, si A est en cours d'exécution au moment où le gestionnaire de transaction amorce la transaction B alors l'ordonnanceur va bloquer B en attendant que A finisse son exécution.

Les actions appliquées à la mémoire, au fur et à mesure que l'exécution continue, forment ainsi une séquence d'actions de type r , w , a ou c que nous appelons ordonnancement. Cet ordonnancement représente un entrelacement possible des opérations des différentes transactions qui ont été soumises au gestionnaire de transactions par les différents *threads* parallèles. Cette appellation vient du fait que les séquences d'actions générées sont dépendantes de l'ordonnanceur implémenté au sein du gestionnaire de transactions.

La question à laquelle nous nous intéressons dans ce chapitre est : dans quelle mesure cet ordonnancement des opérations généré par l'ordonnanceur est-il considéré comme correct ? Il est clair que si l'ordonnanceur empêche une transaction

de s'exécuter une fois que la transaction courante a fini son exécution, alors l'ordonnancement séquentiel généré est considéré comme correct, puisqu'il respecte la sémantique d'exécution du modèle transactionnel représenté plus haut (voir Figure 4.1). Par conséquent, nous utilisons les ordonnancements séquentiels comme référence de validité pour définir les différentes notions de sérialisabilité que nous présentons dans les prochaines sections.

4.2 Sérialisabilité

Dans cette section nous allons définir plusieurs notions de sérialisabilité afin de pouvoir valider qu'une exécution concurrente des transactions du modèle transactionnel présenté ci-dessus est correcte. Pour cela, nous commençons par illustrer trois exemples typiques montrant des situations erronées à la suite d'une exécution concurrente des transactions. Sur la base de ces exemples, nous allons définir les notions de validité afin qu'elles évitent ces situations dangereuses.

Exemple 4.1 *Problème du m-à-j perdue*

Le problème suivant est connu sous le nom de mise à jour perdue, noté "m-à-j perdue". Considérons deux transactions qui s'exécutent concurremment comme suit :

t_1	<i>Temps</i>	t_2
	$/* x = 100 */$	
$r(x)$	1	
	2	$r(x)$
$/* m-à-j x := x + 30 */$	3	
	4	$/* m-à-j x := x + 20 */$
$w(x)$	5	
	$/* x = 130 */$	
	6	$w(x)$
	$/* x = 120 */$	

↑
m-à-j "perdue"

Nous supposons que x est une variable numérique partagée contenant initialement la valeur 100. Les deux transactions t_1 et t_2 lisent la valeur de x dans une variable locale et mettent à jour cette valeur dans la même variable locale. Ensuite, elles écrivent les nouvelles valeurs calculées dans la variable partagée x . Au temps 1 et 2 les transactions t_1 et t_2 lisent la valeur initiale 100 de x . t_1 met à jour cette valeur et écrit la valeur 130 dans x au temps 5. Au temps 6, t_2 met à jour x par 120. En ce moment, la mise à jour de t_1 est perdue et ainsi l'atomicité de l'exécution n'est plus respectée. En effet, une exécution séquentielle des deux transactions aurait fini avec une valeur de x égale à 150. En fait, ce problème surgit à cause de l'agencement des opérations entrelacées de lecture et d'écriture des deux transactions :

$$r_1(x)r_2(x)w_1(x)w_2(x)$$

où le temps progresse de gauche à droite.



Un deuxième problème qui peut surgir à la suite d'une exécution concurrente des transactions est le problème de "lecture inconsistante" illustré par l'exemple 4.2.

Exemple 4.2 *Problème de lecture inconsistante*

Considérons deux transactions t_1 et t_2 qui s'exécutent d'une manière concurrente.

t_1	Temps	t_2
	1	$r(x)$
	2	$/* x := x - 10 */$
	3	$w(x)$
$/* sum := 0 */$	4	
$r(x)$	5	
$r(y)$	6	
$/* sum := sum + x */$	7	
$/* sum := sum + y */$	8	
	9	$r(y)$
	10	$/* y := y + 10 */$
	11	$w(y)$

Supposons que les deux variables partagées, x et y , représentent deux comptes bancaires contenant initialement 50\$. La transaction t_1 fait la somme des deux comptes alors que t_2 fait un transfert d'un compte à un autre. Au temps 3, t_2 met à jour le compte x en lui soustrayant 10\$. Entre temps, t_1 lit les valeurs de x et y , puis elle fait la somme 40\$+50\$ et retourne la valeur 90\$. Dans ce cas, nous sommes en présence d'une situation de lecture inconsistante puisque t_1 n'a pu voir qu'une partie du transfert. Remarquer que, comme pour le problème du "m-à-j perdue", ce problème surgit aussi à cause de l'agencement des opérations entrelacées de lecture et d'écriture des deux transactions :

$$r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)$$



Dans le dernier exemple, nous présentons le problème qui peut se présenter en cas où l'une des transactions est annulée.

Exemple 4.3 *Problème de lecture erroné («dirty-read»)*

Ce problème, connu sous le nom de problème de lecture erronée ou problème de lecture de données non validées, survient lorsqu'une transaction a été annulée

après qu'elle a modifié une donnée qui a été lue par une autre transaction entre le moment de l'écriture et le moment d'annulation.

t_1	<i>Temps</i>	t_2
$r(x)$	1	
$/^* x := x - 100 */$	2	
$w(x)$	3	
	4	$r(x)$
	5	$/^* x := x + 100 */$
annulation	6	
	7	$w(x)$

Supposons que la variable x représente un compte bancaire contenant initialement 100\$. Lorsque la transaction t_1 est annulée après que la valeur modifiée de x a été lue par t_2 , la valeur lue de x devient invalide. Dans ce cas, des mesures nécessaires doivent être prises pour empêcher t_2 d'utiliser cette valeur invalide. Autrement, nous allons nous retrouver avec un compte qui n'a pas changé de valeur alors qu'il y a un dépôt de 100\$, qui vient d'être effectué.



Ce que nous pouvons remarquer des exemples ci-dessus est que des accès concurrents aux données partagées effectués par deux transactions peuvent être conflictuels et causer des effets incorrects. Par conséquent, une certaine forme de contrôle de concurrence est nécessaire.

Une autre observation importante est que les résultats incorrects obtenus sont dus aux agencements des opérations de lecture et écriture. Nous allons montrer, dans les prochaines sections, comment ces agencements vont nous permettre de définir les critères de validité d'une exécution concurrente en distinguant les agencements admissibles de ceux qui pourraient conduire à des anomalies.

4.3 Syntaxe d'un ordonnancement

Avant de procéder à la définition des notions de validité, nous introduisons d'abord la syntaxe d'un ordonnancement et des transactions. Pour cela, nous essayons de respecter la situation dynamique d'un ordonnanceur selon le modèle

présenté à la section 4.1. Ainsi un ordonnancement, en plus des opérations de lecture et écriture, contient les informations déterminant comment une transaction finit son exécution :

- $\text{commit}(c)$: La transaction a terminé son exécution avec succès. En ce moment, l'ordonnanceur libère toutes les informations reliées à la transaction et celle-ci ne peut plus être abandonnée.
- $\text{abort}(a)$: La transaction est abandonnée. Toutes les opérations effectuées jusqu'à ce point doivent être annulées. Remarquer que cette opération peut être activée pour deux raisons :
 - La transaction veut s'annuler par son propre gré.
 - Pour des raisons de validité, l'ordonnanceur doit abandonner la transaction.

Chaque opération dans un ordonnancement s est indexée par un numéro i représentant la transaction t_i qui a exécuté l'opération. Une transaction t_i correspond à la séquence finie d'opérations, de la forme $w_i(x)$ ou $r_i(x)$, obtenue à partir de s en enlevant toutes les opérations des autres transactions ainsi que l'opération de terminaison : a_i ou c_i . Compte tenu de la sémantique d'une transaction qui suppose qu'elle est atomique, sans perte de généralité, nous considérons les hypothèses suivantes lors d'une exécution d'une transaction :

- Dans chaque transaction une entité x est lue ou écrite au plus une fois.
- Aucune entité n'est lue après qu'elle est écrite.

Nous introduisons aussi les notations suivantes pour faciliter la distinction entre les transactions actives, terminées et abandonnées.

- $\text{trans}(s) = \{t_i | s \text{ contient des opérations de } t_i\}$. Il contient toutes les transactions qui ont lieu partiellement ou complètement dans un ordonnancement.
- $\text{commit}(s) = \{t_i \in \text{trans}(s) | c_i \in s\}$

- $\text{abort}(s) = \{t_i \in \text{trans}(s) \mid a_i \in s\}$
- $\text{active}(s) = \text{trans}(s) - (\text{commit}(s) \cup \text{abort}(s))$

Exemple 4.4

Soit l'ordonnancement s suivant :

$$s = r_1(x)r_2(z)r_3(x)w_2(x)w_1(x)r_3(y)r_1(y)w_1(y)w_2(z)w_3(z)c_1c_2a_3$$

Nous obtenons ainsi :

- $\text{trans}(s) = \{t_1, t_2, t_3\}$ tel que

$$t_1 = r(x)w(x)r(y)w(y)$$

$$t_2 = r(z)w(x)w(z)$$

$$t_3 = r(x)r(y)w(z)$$

- $\text{commit}(s) = \{t_1, t_2\}$
- $\text{abort}(s) = \{t_3\}$
- $\text{active}(s) = \{\}$



Nous appelons un ordonnancement s un ordonnancement complet si s ne contient pas des transactions actives, soit :

- $\text{active}(s) = \{\}$, ou
- $\text{trans}(s) = \text{commit}(s) \cup \text{abort}(s)$

4.4 Sémantique d'un ordonnancement

Selon la représentation d'une transaction comme étant une séquence d'opérations de lecture et écriture, une transaction est purement une entité syntaxique

dont la sémantique est inconnue. Si nous avons des informations sur le programme qui exécute une transaction et les transformations d'états voulues des données sous-jacentes, ces connaissances auraient pu être utilisées pour interpréter une transaction et associer une sémantique formelle à celle-ci. En l'absence de cette information, le meilleur que nous pouvons faire est de donner une interprétation syntaxique des transactions et des ordonnancements qui en découlent, qui est la plus générale que possible. Ainsi nous supposons les deux hypothèses suivantes pour un ordonnancement quelconque s :

1. Une opération $r_i(x) \in s$ d'une transaction $t_i \in \text{trans}(s)$ lit la valeur écrite par la dernière opération d'écriture $w_j(x) \in s$, $j \neq i$, qui se produit avant $r_i(x)$.
2. Une opération d'écriture $w_i(x)$ écrit une nouvelle valeur qui dépend potentiellement de toutes les valeurs que t_i a lu de la mémoire ou des autres transactions dans $\text{active}(s_{\leq w_j}) \cup \text{commit}(s_{\leq w_i})$, $s_{\leq w_i}$ est le préfixe de s s'arrêtant à l'opération $w_i(x)$.

Pour faciliter le raisonnement, nous ignorons les transactions abandonnées en enlevant d'un ordonnancement toutes les opérations reliées aux transactions annulées. Nous supposons au préalable que l'ordonnancement prend en charge le problème de lecture erronée, illustré dans l'exemple 4.3, en empêchant, par exemple, l'ordonnancement $r_1(x)w_1(x)r_2(x)a_1w_2(x)c_2$ généré à la suite de ce problème d'avoir lieu. Autrement le fait d'ignorer simplement les transactions abandonnées, cet ordonnancement se réduit à $r_2(x)w_2(x)c_2$, qui est totalement correct.

Pour la première hypothèse introduite plus haut, un problème peut survenir dans le cas où une opération de lecture n'est pas précédée par une opération d'écriture, comme dans le cas de l'ordonnancement s suivant :

$$s = r_1(x)r_2(y)w_1(x)r_2(x) \dots$$

Pour remédier à ce problème, nous supposons que chaque ordonnancement s est précédé par une transaction fictive t_0 qui écrit dans toutes les données référencées

dans s puis commit. Ainsi t_0 joue le rôle d'une transaction d'initialisation qui définit une valeur initiale à toutes les données référencées dans un ordonnancement. Nous supposons aussi qu'un ordonnancement a non seulement une transaction d'initialisation t_0 mais aussi une transaction t_∞ qui s'exécute complètement après toutes les transactions et qui lit toutes les données écrites. Par exemple, l'ordonnancement juste mentionné plus haut devient

$$s = w_0(x)w_0(y)c_0r_1(x)r_2(y)w_1(x)r_2(x) \dots r_\infty(x)r_\infty(y)c_\infty$$

Le problème avec la sémantique informelle des opérations d'un ordonnancement définie plus haut est que les valeurs lues et écrites ne sont pas connues. Une façon pour remédier à cette situation est d'utiliser la sémantique de *Herbrand* qui donne une interprétation symbolique aux valeurs lues et écrites par le biais de symboles de fonctions non interprétés.

Définition 4.5 *Sémantique de Herbrand des opérations*

Soit s un ordonnancement. La sémantique de *Herbrand*, notée H_s , des opérations $r_i(x)$ et $w_i(x) \in op(s)$ est récursivement définie comme suit :

1. $H_s(r_i(x)) = H_s(w_j(x))$, où $w_j(x)$, $j \neq i$, est la dernière opération d'écriture de x dans s avant $r_i(x)$.
2. $H_s(w_i(x)) = f_{ix}(H_s(r_i(y_1)), \dots, H_s(r_i(y_m)))$, où $r_i(y_j)$, $1 \leq j \leq m$, représentent toutes les opérations de lecture de t_i qui se produisent avant $w_i(x)$, et où f_{ix} est un symbole de fonction non interprété d'arité m .

◆

Noter que le symbole de fonction f_{ix} est bien défini dans la définition 4.5 grâce à l'hypothèse qu'il ne peut y avoir qu'au maximum une opération d'écriture dans chaque donnée pour chaque transaction.

Comme exemple, considérons l'ordonnancement s suivant :

$$s = w_0(x)w_0(y)c_0r_1(x)r_2(y)w_2(x)w_1(y)c_2c_1r_\infty(x)r_\infty(y)c_\infty$$

La sémantique de *Herbrand*, H_s , de chaque opération de s est la suivante :

$$\begin{aligned} H_s(w_0(x)) &= f_{0x}() \\ H_s(w_0(y)) &= f_{0y}() \\ H_s(r_1(x)) &= H_s(w_0(x)) = f_{0x}() \\ H_s(r_2(y)) &= H_s(w_0(y)) = f_{0y}() \\ H_s(w_2(x)) &= f_{2x}(H_s(r_2(y))) = f_{2x}(f_{0y}()) \\ H_s(w_1(y)) &= f_{1y}(H_s(r_1(x))) = f_{1y}(f_{0x}()) \\ H_s(r_\infty(x)) &= H_s(w_2(x)) = f_{2x}(f_{0y}()) \\ H_s(r_\infty(y)) &= H_s(w_1(y)) = f_{1y}(f_{0x}()) \end{aligned}$$

Nous généralisons maintenant la sémantique de *Herbrand* pour un ordonnancement s .

Définition 4.6 *Sémantique de Herbrand d'un ordonnancement*

La sémantique de *Herbrand* d'un ordonnancement s , notée $H[s]$, est la fonction qui attribue à chaque donnée x sa valeur $H[s](x)$ définie comme suit

$$H[s](x) = H_s(r_\infty(x))$$



En d'autres termes, la sémantique d'un ordonnancement est l'ensemble des valeurs qui ont été dernièrement écrites. Prenons le même ordonnancement s vu juste plus

haut :

$$s = w_0(x)w_0(y)c_0r_1(x)r_2(y)w_2(x)w_1(y)c_2c_1r_\infty(x)r_\infty(y)c_\infty$$

Nous obtenons ainsi :

$$H[s](x) = H_s(r_\infty(x)) = f_{2x}(f_{0y}())$$

$$H[s](y) = H_s(r_\infty(y)) = f_{1y}(f_{0x}())$$

Nous sommes maintenant prêts pour définir une première notion de validité.

4.5 Sérialisabilité “Final State”

Définition 4.7 *Équivalence “Final State”*

Soit s et s' deux ordonnancements. s et s' sont dits équivalents “Final State”, que nous notons $s \approx_f s'$, si

- $\text{op}(s) = \text{op}(s')$ et
- $H[s] = H[s']$



Intuitivement, deux ordonnancements sont équivalents “Final State” s’ils finissent avec le même état final pour n’importe quel état initial. Par exemple, considérons les deux ordonnancements suivants :

$$s = r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x)$$

$$s' = r_3(z)w_3(z)r_2(y)r_2(x)w_2(z)r_1(x)w_1(y)w_1(x)$$

Nous obtenons alors :

$$\begin{aligned} H[s](x) &= Hs(w_1(x)) = f_{1x}(f_{0x}()) = H_{s'}(w_1(x)) = H[s'](x) \\ H[s](y) &= Hs(w_1(y)) = f_{1y}(f_{0x}()) = H_{s'}(w_1(y)) = H[s'](y) \\ H[s](z) &= Hs(w_2(z)) = f_{2z}(f_{0x}(), f_{0y}()) = H_{s'}(w_2(z)) = H[s'](z) \end{aligned}$$

Ainsi $s \approx_f s'$. Ensuite, considérons les deux ordonnancements complets suivants :

$$\begin{aligned} s &= r_1(x)r_2(y)w_1(y)w_2(y)c_1c_2 \\ s' &= r_1(x)w_1(y)r_2(y)w_2(y)c_1c_2 \end{aligned}$$

Nous obtenons alors :

$$\begin{aligned} H[s](y) &= H_s(w_2(y)) = f_{2y}(f_{0y}()) \\ H[s'](y) &= H_{s'}(w_2(y)) \\ &= f_{2y}(H_{s'}(r_2(y))) \\ &= f_{2y}(H_{s'}(w_1(y))) \\ &= f_{2y}(f_{1y}(H_{s'}(r_1(x)))) \\ &= f_{2y}(f_{1y}(f_{0x}())) \end{aligned}$$

Ainsi, $s \not\approx_f s'$. Cet exemple en particulier montre que l'équivalence "Final State" ne peut pas être décidée seulement en regardant la dernière opération d'écriture effectuée. Tout ce qui précède cette écriture doit être pris en compte. En effet, dans s , la valeur finale de y écrite par t_2 dépend de l'ancienne valeur de y lue auparavant par t_2 . D'un autre côté, dans s' , la valeur de y écrite par t_1 influence la valeur finale de celle-ci écrite par t_2 , alors que t_1 est sans effet dans s .

Maintenant, étant donnée une notion d'équivalence, nous pouvons définir une première notion de validité basée sur l'équivalence d'un ordonnancement à un ordonnancement séquentiel.

Définition 4.8 *Sérialisabilité* “Final State”

Un ordonnancement complet s est sérialisable “Final State” s'il existe un ordonnancement séquentiel s' tel que $s \approx_f s'$.

FSR dénote la classe de tous les ordonnancements sérialisables “Final State”. ◆

Nous revenons aux deux anomalies “m-à-j perdue” et “lecture inconsistante” présentées respectivement dans les exemples 4.1 et 4.2 afin de vérifier si un ordonnancement $s \in \text{FSR}$ permet de les éviter. Il est clair que le problème de “m-à-j perdue” est classé par FSR parmi les comportements inadmissibles. En effet, l'ordonnancement de l'exemple 4.1

$$r_1(x)r_2(x)w_1(x)w_2(x)c1c2$$

n'appartient pas à la classe FSR. Cependant l'exemple de la lecture inconsistante montre que FSR est encore insuffisante comme critère de validité puisque l'ordonnancement

$$r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)c1c2$$

est accepté selon le critère de validité FSR étant donné que l'état final est le même que produit l'ordonnancement séquentiel t_2t_1 . Le critère de validité *Sérialisabilité* “View”, présentée dans la prochaine section, permettra de prévenir le problème de lecture inconsistante.

4.6 Sérialisabilité “View”

Définition 4.9 *Équivalence et Sérialisabilité* “View”

-
1. Soit s et s' deux ordonnancements. s et s' sont dits équivalents “View”, que nous notons $s \approx_v s'$, si
 - (a) $\text{op}(s) = \text{op}(s')$ et
 - (b) $H[s] = H[s']$ et
 - (c) $H_s(p) = H_{s'}(p)$ pour toute opération p d'écriture ou de lecture.
 2. Un ordonnancement complet s est sérialisable “View” s'il existe un ordonnancement séquentiel s' tel que $s \approx_v s'$.

VSR dénote la classe de tous les ordonnancements sérialisables “View”.



La condition (a) et (b) de la définition 4.9.(1) correspondent à la définition de FSR. La condition (c) nécessite en plus que la sémantique de chaque opération dans deux ordonnancements équivalents “View” soit la même. Le résultat de ceci est que l'ordonnancement suivant représentant le problème de lecture inconsistante

$$r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)c1c2$$

n'est plus admissible puisqu'il n'est pas équivalent ni à l'ordonnancement séquentiel $t_1 t_2$ ni à $t_2 t_1$.

Nous pouvons conclure ainsi que VSR est un critère de validité acceptable afin de décider si un ordonnancement est correct ou non puisqu'elle nous permet d'éviter les deux problèmes typiques que peut engendrer une exécution concurrente de plusieurs transactions. Cependant au point de vue pratique, VSR exhibe encore certaines défaillances qui l'empêche d'être adopté par un ordonnanceur qui s'exécute dans un environnement dynamique. En effet, décider si un ordonnancement est dans VSR est un problème NP complet. En plus, l'efficacité n'est pas la seule raison qui rend VSR peu appropriée comme critère de validité. Pour montrer ceci, il nous faut d'abord la définition de monotonie qui suit. Soit s un ordonnancement et $T \subseteq \text{trans}(s)$. $\Pi_T(s)$ dénote la projection de s sur T , c'est à dire, l'ordonnancement s' avec les opérations $\text{op}(s') = \text{op}(s) - \bigcup_{t \notin T} \text{op}(s)$, obtenu

en enlevant de s toutes les opérations qui ne sont pas dans T . Par exemple, si

$$s = w_1(x)r_2(x)w_2(y)r_1(y)w_1(y)w_3(x)w_3(y)c_1a_2$$

et $T = \{t_1, t_2\}$, alors

$$\Pi_T(s) = w_1(x)r_2(x)w_2(y)r_1(y)w_1(y)c_1a_2$$

Définition 4.10 *Monotonicit *

Une classe E d'ordonnements est dite monotone si : si $s \in E$, alors $\Pi_T(s) \in E$ pour chaque $T \subseteq \text{trans}(s)$. En d'autres termes, E est ferm  sous n'importe quelle projection.



La monotonicit  d'une classe E est une propri t  d sirable puisqu'elle pr serve E sous n'importe quelle projection. Dit autrement, si dans une situation d'ordonnement dynamique, un ordonnement s n'appartient plus   une classe E monotone, il n'est plus n cessaire de traiter s davantage et l' tendre avec d'autres op rations puisque nous pouvons savoir, gr ce   la monotonicit  de E , que quelque soit l'ordonnement qui en r sulte, il ne peut pas appartenir   E .

Si nous consid rons la classe VSR, nous pouvons v rifier facilement que VSR n'est pas monotone. Pour cela, soit s l'ordonnement suivant :

$$s = w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1w_3(x)w_3(y)c_3$$

Ainsi nous avons $s \in \text{VSR}$ puisque $s \approx_v t_1t_2t_3 \approx_v t_2t_1t_3$. Cependant,

$$\Pi_{\{t_1, t_2\}} = w_1(x)w_2(x)w_2(y)c_2w_1(y)c_1 \notin \text{VSR}$$

Intuitivement, dans cet exemple, la derni re transaction t_3 dissimule tout effet d    l'ex cution pr c dente entrelac e de t_1 et t_2 . Ainsi, si nous enlevons t_3 par

projection, l'ordonnancement résultant n'est plus valide par rapport à VSR. Par conséquent, VSR est encore insuffisante comme critère de validité, ce qui nous emmène à une nouvelle notion de validité plus restrictive.

4.7 Sérialisabilité conflictuelle

La prochaine notion de validité, qui est la sérialisabilité conflictuelle, constitue la notion la plus pratique pour les systèmes d'information transactionnelle, particulièrement pour concevoir des ordonnanceurs. Comme nous allons le voir, elle est rapide à tester du point de vue de l'efficacité algorithmique et ainsi elle diffère considérablement des notions de sérialisabilité discutées plus haut.

La sérialisabilité conflictuelle est basée sur une notion simple de conflit définie comme suit.

Définition 4.11 *opérations conflictuelles et relation conflictuelle*

Soit s un ordonnancement, t et $t' \in \text{trans}(s)$ et $t \neq t'$.

1. deux opérations $p \in t$ et $q \in t'$ sont en conflit, ou conflictuelles, si elles accèdent à une même entité x et l'une des opérations est une opération d'écriture, soit :

$$(p = r(x) \wedge q = w(x)) \vee (p = w(x) \wedge q = r(x)) \vee (p = w(x) \wedge q = w(x))$$

2. $\text{conf}(s) = \{(p, q) | p, q \text{ sont conflictuelles dans } s \text{ et } p <_s q\}$ est appelée la relation conflictuelle de s .



Remarquer que la relation conflictuelle ne prend pas les opérations de terminaison c et a en considération. Si une opération d'abandon a est présente dans un ordonnancement alors il suffit de l'ignorer et les conflits avec les autres transactions

sont omis de $\text{conf}(s)$. Par exemple, si

$$s = w_1(x)r_2(x)w_2(y)r_1(y)w_1(y)w_3(x)w_3(y)c_1a_2$$

où t_1 est validée, t_2 abandonnée et t_3 est encore active, nous aurons

$$\text{conf}(s) = \{(w_1(x), w_3(x)), (r_1(y), w_3(y)), (w_1(y), w_3(y))\}$$

Maintenant nous sommes prêts de définir les notions d'équivalence et de sérialisabilité suivantes.

Définition 4.12 *Équivalence et Sérialisabilité conflictuelle*

1. Soit s et s' deux ordonnancements. s et s' sont équivalents “conflictuels”, que nous notons $s \approx_c s'$, si les deux conditions suivantes sont satisfaites :
 - (a) $\text{op}(s) = \text{op}(s')$ et
 - (b) $\text{conf}(s) = \text{conf}(s')$
2. Un ordonnancement complet s est “sérialisable conflictuel” s’il existe un ordonnancement s' tel que $s \approx_c s'$.

CSR dénote la classe de tous les ordonnancements “sérialisables conflictuels”.



Le théorème suivant montre la relation entre les trois classes de sérialisabilité décrites plus haut.

Théorème 4.13

$$\text{CSR} \subset \text{VSR} \subset \text{FSR}$$



Pour l'inclusion stricte, $VSR \subset FSR$, nous avons déjà vu comment l'ordonnement dû au problème de lecture inconstante est dans FSR et non VSR . L'ordonnement s suivant illustre le cas de la deuxième inclusion puisqu'il appartient à $VSR(s \approx_v t_1 t_2 t_3)$ et non CSR .

$$s = w_1(x)w_2(x)w_2(y)w_1(y)w_3(x)w_3(y)$$

La différence importante entre CSR et les deux notions de sérialisabilité FSR et VSR est que l'appartenance à la première peut être vérifiée efficacement. Pour cela nous allons caractériser CSR en terme de théorie de graphe à l'aide du graphe de sérialisabilité défini comme suit.

Définition 4.14 *Graphe de sérialisabilité*

Soit s un ordonnancement. Le graphe de sérialisabilité, $G(s) = (V, E)$ de s , est défini par :

- $V = \text{commit}(s)$.
- $(t, t') \in E \iff t \neq t' \wedge (\exists p \in t)(\exists q \in t').(p, q) \in \text{conf}(s)$



Le théorème suivant est largement connu sous le nom de théorème de sérialisabilité conflictuelle.

Théorème 4.15 *Théorème de sérialisabilité*

Soit s un ordonnancement.

$$s \in CSR \iff G(s) \text{ est acyclique.}$$



D'où le corollaire suivant.

Corollaire 4.16

L'appartenance à la classe CSR peut être testée en temps polynomial au nombre de transactions pour un ordonnancement donné.

Démonstration. Suite au théorème de sérialisabilité, le graphe de sérialisabilité peut être construit en un temps linéaire au nombre d'opérations dans un ordonnancement donné, et le test de l'existence d'un cycle peut être fait en un temps au maximum quadratique au nombre de nœuds de ce graphe. \square



Voici un exemple qui illustre des graphes de sérialisabilité.

Exemple 4.17

Considérons les deux ordonnancements suivants.

- $s = r_1(y)r_3(w)r_2(y)w_1(y)w_1(x)w_2(x)w_2(z)w_3(x)c_1c_3c_2$
- $s' = r_1(x)r_2(x)w_2(y)w_1(x)c_2c_1$

Les graphes de sérialisabilité $G(s)$ et $G(s')$ correspondant respectivement à s et s' sont représentés à la Figure 4.3.

Puisque $G(s)$ contient un cycle alors $s \notin \text{CSR}$, et puisque $G(s)$ est acyclique alors $s' \in \text{CSR}$.



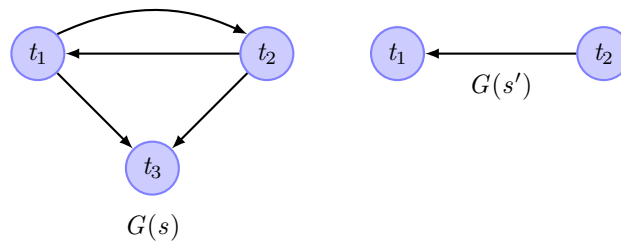


Figure 4.3 Graphes de sérialisabilité de l'exemple 4.17.

4.8 Stratégies d'ordonnement

Le fait d'utiliser la notion de transaction pour la conception des systèmes sur puce procure deux autres avantages clés, contrairement aux autres disciplines telles que les systèmes de gestion des bases de données ou les mémoires transactionnelles (voir section 3.1) :

- Les transactions qui constituent les modèles transactionnels sont déjà connues et prédéfinies. En conséquence, ceci va nous permettre d'effectuer une analyse statique préalable afin de déterminer les opérations de lecture et écriture et les relations de conflits entre les transactions.
- Contrairement aux systèmes de bases de données dans lesquels les transactions arrivent arbitrairement et dépendent de l'environnement extérieur, le simulateur d'un modèle transactionnel peut décider comment les transactions sont allouées aux *threads* et quand elles peuvent s'exécuter et dans quel ordre. L'essentiel est que l'ordonnement généré soit dans CSR.

Dans ce qui suit, nous allons expliquer comment nous pouvons utiliser ces deux avantages pour concevoir un simulateur de modèle transactionnel.

Un résultat important de l'analyse des transactions est la construction de multigraphe de conflit et de graphe de conflit dont les définitions suivent.

Définition 4.18 *Multigraphe et Graphe de conflit*

Un multigraphe de conflit $M(T)$ d'un ensemble de transactions T est défini comme suit.

1. Chaque nœud du graphe correspond à une transaction.
2. Chaque arête du graphe correspond à deux opérations conflictuelles selon la définition 4.11

Nous appelons graphe de conflit le graphe obtenu à partir du multigraphe de conflit en fusionnant toutes les arêtes parallèles liant chaque paire de nœuds.



Un exemple de multigraphe de conflit est représenté dans la Figure 4.4. Chaque arête est étiquetée par la variable partagée impliquée dans le conflit.

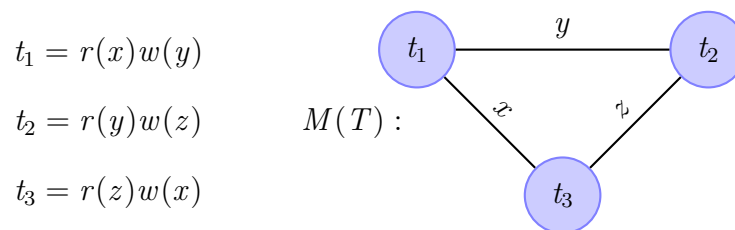


Figure 4.4 Exemple de Multigraphe de conflit.

Puisque nous pouvons avoir plusieurs conflits entre deux transactions, il peut y avoir plusieurs arêtes parallèles entre les nœuds correspondants à ces transactions. Lors de l'exécution de ces transactions, plusieurs arêtes vont être regroupées en un seul arc dans le graphe de sérialisabilité correspondant à cette exécution, à condition que les conflits soient résolus dans la même direction. Ainsi, si $G(s)$ est le graphe de sérialisabilité pour un ordonnancement s des transactions dans T , chaque arc de $G(s)$ correspond à une ou plusieurs arêtes de $M(T)$. Si $G(s)$ possède un circuit, alors $M(T)$ a un cycle impliquant les mêmes transactions que ce circuit. En général pour chaque circuit dans $G(s)$ correspond un cycle dans

$M(T)$. Inversement, si $M(T)$ est acyclique alors $G(s)$ est acyclique aussi, et ceci pour tout ordonnancement s de l'ensemble des transactions T .

L'exemple de la figure 4.5 montre la correspondance entre un multigraphe de conflit et un graphe de sérialisabilité pour un ordonnancement particulier. Chaque arête du multigraphe est étiquetée par la paire d'opérations conflictuelles. Puisque nous avons trois paires d'opérations conflictuelles, toutes liées à la même variable x , le multigraphe contient trois arêtes liant les mêmes nœuds. Lors de l'exécution de ces deux transactions les trois arêtes vont correspondre soit à un circuit, soit à un arc orienté de t_1 à t_2 , ou soit à un arc orienté de t_2 à t_1 dans le graphe de conflit correspondant. La Figure 4.5 représente ces trois cas pour trois ordonnancements possibles.

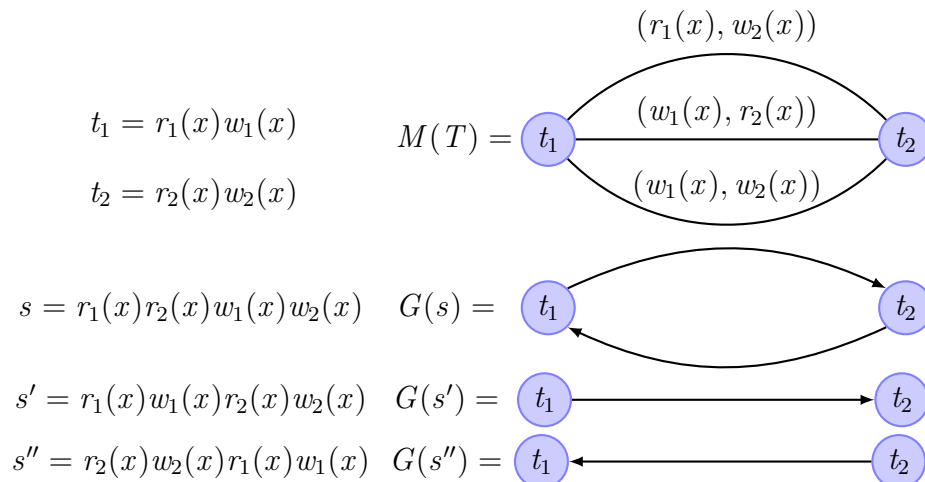


Figure 4.5 Correspondance entre multigraphe de conflit et graphe de sérialisabilité.

À la suite de cette discussion expliquant la correspondance entre un graphe de sérialisabilité et un multigraphe de conflit, puis en se basant sur le théorème de sérialisabilité 4.15 qui dit que pour qu'une exécution des transactions soit dans CSR, il faut que le graphe de sérialisabilité de l'ordonnancement associé à cette exécution soit acyclique, nous allons utiliser les stratégies qui suivront pour concevoir notre simulateur.

En comparaison avec l'exécution séquentielle de la figure 4.1 qui choisit une transaction à chaque itération, le simulateur parallèle, durant chaque itération, ordonnance toutes les transactions du modèle transactionnel en utilisant différentes techniques et mécanismes de synchronisation (voir figure 4.6).

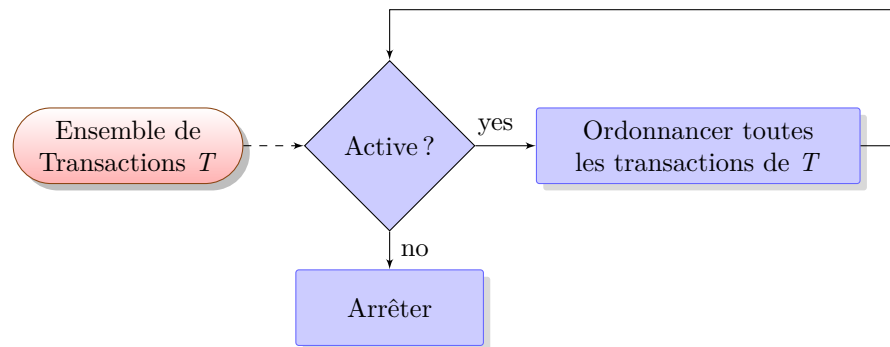


Figure 4.6 Exécution Concurrente

Ces techniques d'ordonnancement reposent sur deux stratégies principales afin de balancer entre parallélisme et surcoût de synchronisation :

1. Deux transactions conflictuelles (ayant une arête commune dans le graphe de conflit) ne doivent pas s'exécuter en parallèle. Ceci permet de lancer les transactions non conflictuelles en parallèle et évite d'ajouter des mécanismes de synchronisation au sein des transactions en garantissant que le graphe de sérialisabilité généré est toujours acyclique. Voir la section 5.3 pour plus de détails sur cette stratégie et les différentes techniques utilisées pour l'implémenter.
2. Lancer toutes les transactions en parallèle. Empêcher de se retrouver avec un graphe de sérialisabilité cyclique en ajoutant des verrous à certaines transactions choisies selon des critères bien définis. Les détails de cette stratégie sont décrits dans la sous-section 6.2.4. L'avantage de cette stratégie est qu'elle permet plus de parallélisme. Par exemple, si le multigraphe est acyclique, alors les transactions peuvent s'exécuter en parallèle sans besoin de les synchroniser. Cependant en cas où le nombre de cycles est important,

il est probable qu'il ne va pas y avoir assez de transactions qui peuvent se dérouler en parallèle et le surcoût dû à la synchronisation peut dégrader les performances en comparaison avec la première stratégie.

Le chapitre qui suivra va détailler les techniques et les outils utilisés pour le développement de l'environnement de simulation. Le chapitre d'après évalue les différentes stratégies d'ordonnancement implémentées et les optimisations qui ont été apportées.

Environnement de modélisation et simulation¹

La méthodologie adoptée pour la conception des systèmes en se basant sur les transactions fait la distinction entre un modèle de transaction décrivant simplement la structure et la fonctionnalité du système cible et un modèle exécutable permettant de simuler la fonctionnalité de ce système. Pour cela, nous avons développé un compilateur qui permet de générer un modèle exécutable à partir d'un modèle de transaction (dans la même optique, nous pouvons cibler la génération d'un modèle Bluespec ou directement un modèle VHDL). Ainsi, une fois qu'un modèle transactionnel, décrivant un système microélectronique, est défini, il passe par plusieurs phases de compilation dans le but de générer un programme exécutable et parallèle prêt pour la simulation et la validation dynamique et qui prend avantage d'une machine multicœur dans le but d'accélérer le temps de simulation. Ces phases de compilation sont automatiques et ne nécessitent aucune intervention de l'utilisateur. Voici une description de chaque phase spécifiée dans la Figure 5.1.

- **Compilateur .NET** : Il s'agit d'un des compilateurs standards qui est fourni avec la plateforme .NET. Ce compilateur va générer un modèle transactionnel en code binaire CIL. Ainsi toutes les phases de compilation qui suivront vont être basées sur ce code intermédiaire du modèle initial. Ceci nous permet d'être indépendants du langage de programmation et ainsi pouvoir supporter tous les langages disponibles sur la plateforme .NET.

1. Un travail antérieur à celui présenté dans ce chapitre a été publié[49], décrivant un environnement de vérification semi-formelle, mais qui utilise un modèle d'exécution événementiel plutôt que transactionnel.

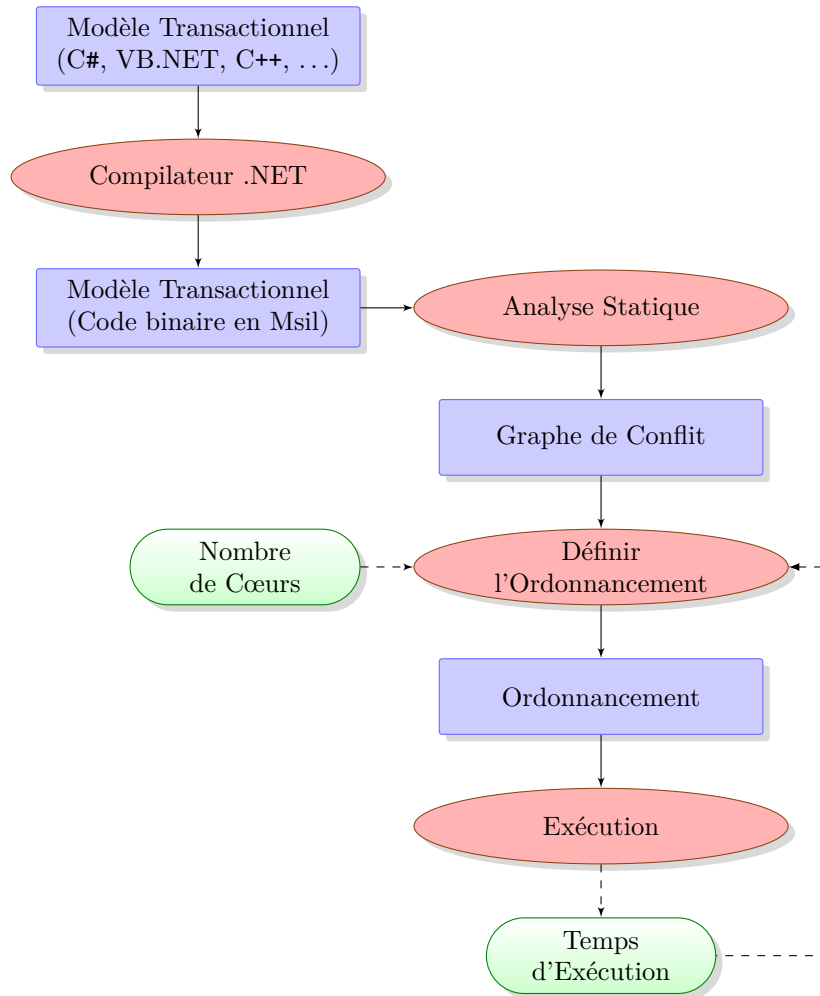


Figure 5.1 Environnement de Modélisation et Simulation

- **Analyse Statique** : Cette phase va analyser les transactions du modèle transactionnel afin de détecter les variables partagées entre elles et ainsi générer le graphe de conflit de ces transactions.
- **Définir l'ordonnancement** : Cette phase va déterminer un ordonnancement d'exécution des transactions. Cet ordonnancement doit respecter l'atomicité de chaque transaction tout en essayant de maximiser la concurrence entre ces transactions. Il dépend aussi du nombre de cœurs de la machine de simulation courante ainsi que d'une estimation moyenne de temps d'exécution de chaque transaction.

- **Exécution** : Cette phase va s'occuper de l'exécution du modèle transactionnel selon l'ordonnancement défini dans la phase précédente. Ainsi elle va être en charge de la création des unités d'exécutions (Threads) et l'allocation des transactions à ces *threads*.

Les prochaines sections donnent plus de détails sur la structure du modèle transactionnel adoptée ainsi que sur les implémentations des différentes phases de compilation définies plus haut.

5.1 Modèle transactionnel

Dans cette section nous allons présenter la structure d'un modèle transactionnel qui est, dans notre cas, le modèle de base pour décrire un système microélectronique. La Figure 5.2 présente les principaux constituants d'un modèle transactionnel.

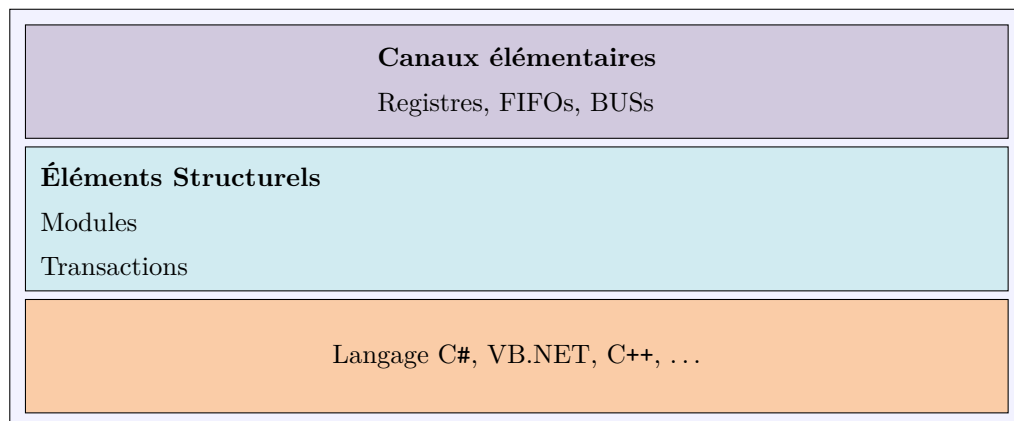


Figure 5.2 Éléments d'un modèle transactionnel.

Voici les éléments de base qui constituent un modèle transactionnel.

Module : Un module est l'unité principale pour décrire un composant. Selon le niveau d'abstraction utilisé, il peut décrire toute la fonction d'un processeur à un niveau d'abstraction algorithmique ou simplement un registre matériel s'il s'agit d'un niveau microarchitecture. Un module est constitué de zéro, une ou plusieurs transactions, expose des méthodes qui peuvent être appelées

par des transactions d'autres modules et spécifie les membres qui définissent l'état du module.

Transaction : C'est l'unité d'exécution pour décrire la fonction désirée. L'ensemble des opérations à l'intérieur d'une transaction est considéré comme atomique et indivisible. La garde d'une transaction est traitée plutôt d'une façon implicite. En effet au lieu que le concepteur définisse une garde au début de chaque transaction, nous avons ajouté une simple construction `Retry`[28] que le concepteur puisse appeler s'il veut annuler la transaction. Conceptuellement, la fonction `Retry` annule la transaction sans laisser d'effets, puis elle relance son exécution depuis le début. Cependant, pour des raisons de performances, différentes implémentations peuvent exister. Par exemple :

- Une implémentation de `Retry` peut bloquer la transaction jusqu'à elle détecte qu'un chemin d'exécution alternatif devient possible.
- Afin d'optimiser le temps de relance, une implémentation de `Retry` peut redémarrer une transaction du point où elle a été bloquée¹ si elle peut valider que l'atomicité de la transaction n'est pas mise en jeu.
- À la suite d'une analyse statique, une implémentation peut déterminer la garde d'une transaction, puis ne l'exécuter que si sa garde est vraie.

Le code de l'exemple 5.1 montre l'utilisation de la méthode `Retry` lors de l'écriture dans une FIFO.

Exemple 5.1 *Utilisation de `Retry` lors d'une écriture dans une FIFO*

```
public void enq(T x) {

    if (num_elements == data_size) Retry();

    data[(head + num_elements) % data_size].write(x);
    num_elements.write(num_elements.read() + 1);
}
```

1. Juste avant la condition qui a causé le blocage de la transaction.

Si la FIFO est pleine(`num_elements == data_size`) alors il faut arrêter l'exécution jusqu'à ce que la condition `num_elements == data_size` devienne fausse. Ceci est spécifié à l'aide de la construction `Retry`. Dans ce cas la condition `num_elements != data_size` est considérée la garde de la méthode `enq`.



Ainsi un modèle transactionnel va être constitué d'un module racine. Celui-ci va déclarer d'autres modules et ces modules vont déclarer à leur tour d'autres modules et ainsi de suite. Nous aurons ainsi une hiérarchie de modules composés d'un ensemble de transactions qui communiquent entre elles. L'Exemple 5.2 présente une description de producteur et consommateur afin de mieux illustrer la structure d'un modèle transactionnel.

Exemple 5.2 *Modèle Producteur-Consommateur*

```
[TopModule]
public class ProducerConsumer : BaseModule {
    Fifo<int> fifo;
    Consumer consumer;
    Producer producer;

    public ProducerConsumer(string[] args) {
        int fifosize = Convert.ToInt32(args[0]);
        fifo = new Fifo<int>(fifosize);
        producer = new Producer(fifo);
        consumer = new Consumer(fifo);
    }
}
```

Dans cet exemple la racine du modèle transactionnel est le module `ProducerConsumer`. Ceci est défini à travers l'attribut `TopModule`. Chaque module racine doit comporter un constructeur avec un tableau de `string` comme argument. Ceci est équivalent à la fonction d'entrée `Main` d'un programme.

Le module `ProducerConsumer` déclare trois autres modules : `producer`, `consumer` et `fifo`. Si nous regardons de près le module `producer`, nous allons voir qu'il est composé d'une transaction :

```
class Producer : BaseModule {
    // déclaration
    [Transaction]
    public void RunProducer() {
        . . .
        value = ComputeValue();
        fifo.enq(value);
    }
}
```

La transaction `RunProducer` décrit la fonctionnalité principale du producteur. De la même manière, le consommateur définit lui aussi une transaction `RunConsumer`. Ainsi notre modèle va être composé de deux transactions qui communiquent à travers une FIFO.



5.2 Analyse statique

La Figure 5.3 montre les différentes étapes exécutées par la phase de l'analyse statique afin de générer le graphe de conflit entre les transactions. Voici la description des différentes étapes.

5.2.1 Élaboration

C'est la phase d'initialisation du modèle transactionnel. C'est pendant cette phase que les modules sont instanciés et les membres sont créés. La phase d'élaboration du modèle transactionnel consiste initialement à appeler le constructeur principal `TopModule(string[] args)` du module racine. Ce dernier va instancier les modules qui le constituent qui eux instancient d'autres modules et ainsi de suite. Ainsi à la fin de l'appel la structure du modèle transitionnel est complètement élaborée. À partir de ce moment, un graphe de relation entre objets est construit

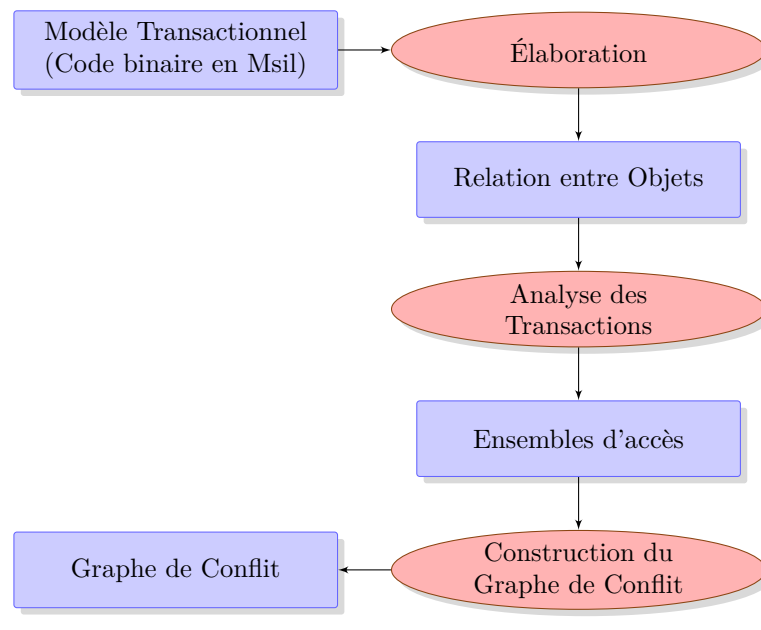


Figure 5.3 Phase de l'Analyse Statique

et ce n'est qu'à ce moment-là que la phase d'analyse statique commence. Le fait d'appliquer l'analyse après l'élaboration du modèle donne certains avantages majeurs que voici :

- Pouvoir définir des modèles transactionnels assez génériques qui peuvent avoir différentes implémentations. Par exemple pour un modèle donné nous pouvons avoir plusieurs architectures possibles : pipeline ou combinatoire, nombre d'étages du pipeline, nombre de blocs parallèles dans chaque étage, etc. Ce n'est qu'à partir des valeurs des arguments du constructeur racine `TopModule(string[] args)` que l'architecture à utiliser est spécifiée et instanciée. Si l'analyse devait commencer avant l'exécution de la phase d'élaboration, il n'aurait pas été possible de déterminer avec précision la structure du modèle ainsi que les différentes transactions qui le composent.
- Le fait de savoir avec précision tous les objets créés ainsi que leur type permet d'être plus précis lors de l'analyse statique et ainsi avoir des informations plus détaillées sur les transactions et les variables partagées. Par exemple, prenons le cas où une fonction virtuelle est appelée. Si l'objet à laquelle

elle appartient n'est pas encore connu alors on ne peut pas déterminer avec précision tous les accès effectués par cette fonction et l'information sur les variables partagées doit être conservative. Ceci pourrait avoir un impact négatif sur le temps de simulation du modèle.

Voici un pseudo-code qui montre les différentes étapes qui sont exécutées lors de cette phase.

```
public void Elaboration(AssemblyNode assemblyNode,
    string[] args) {

    // Chercher le module Racine
    topModule = getTopModule();

    //Appeler le constructeur TopModule(string[] args)
    Assembly assembly = Assembly.LoadFrom(assemblyNode.Location);
    object root = assembly.CreateInstance(topModule.ToString(),
        false, BindingFlags.CreateInstance,
        null, new object[] { args }, null, null);

    // Construire le garaphe de relation entre objets
    ConstructRelationGraph(root);
}
```

L'exemple montre l'avantage de .NET d'utiliser la réflexion afin de pouvoir charger dynamiquement, lors de l'exécution, un assemblage (`Assembly.LoadFrom`) puis de retrouver et instancier un objet (`assembly.CreateInstance`). Nous parlons ici de liaison tardive («*late-binding*») pour décrire la création d'un objet à l'exécution. La fonction `ConstructRelationGraph (root)` permet de parcourir récursivement les membres de chaque objet créé et établir la relation entre `objet.membre` et `objet`. En prenant l'Exemple 5.2 cette relation entre membre et objet va se traduire par le graphe de relation de la Figure 5.4.

5.2.2 Analyse des transactions

Après l'élaboration du modèle transactionnel, le compilateur va parcourir tous les modules instanciés afin de déterminer toutes les transactions déclarées, puis il

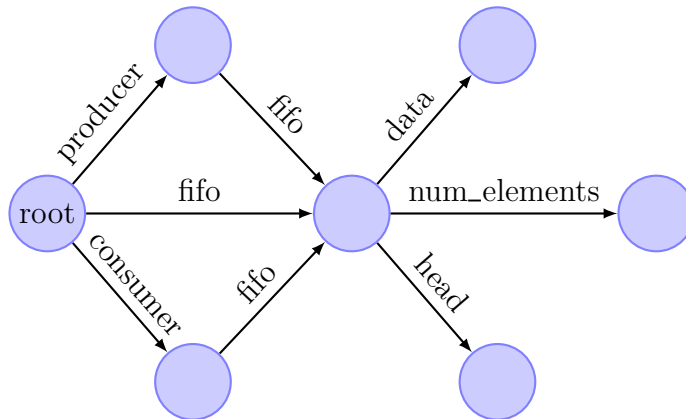


Figure 5.4 Graphe de Relation de l'Exemple 5.2

va les analyser afin de déterminer pour chaque transaction les accès effectués en mode lecture et écriture sur les états du modèle. Dans cette analyse nous avons utilisé une implémentation de l'analyse de pointeurs faite par [11] qui est aussi une adaptation du travail de [53, 54] fait sur Java.

Le résultat de cette étape est de déterminer un ensemble de lecture R_i et un ensemble d'écriture W_i pour chaque transaction T_i . Chaque ensemble R_i contient les accès en lecture qui vont être effectués durant l'exécution de la transaction T_i et chaque ensemble W_i contient les accès en écriture qui vont être effectués durant l'exécution de la transaction T_i . Les éléments de ces ensembles sont des expressions de chemins qui vont être résolus dans la prochaine étape (lors de la construction du graphe de conflit) aux objets réels créés pendant l'étape d'élaboration.

Par exemple, reprenons l'exemple 5.2 du producteur-consommateur. Durant cette étape deux transactions vont être analysées. Une transaction T_c qui correspond à la transaction du consommateur et une transaction T_p qui correspond à la transaction du producteur. Les ensembles de lecture et écriture de T_p vont

contenir les éléments suivants.

$$\begin{aligned}
 R_p &= \{ \textit{this.fifo.head} && , \\
 & \quad \textit{this.fifo.num_elements} && , \\
 & \quad \textit{this.fifo.data_size} && \} \\
 W_p &= \{ \textit{this.fifo.data} && , \\
 & \quad \textit{this.fifo.num_elements} && \}
 \end{aligned}$$

Les ensembles de lecture et écriture de T_c vont contenir les éléments suivants

$$\begin{aligned}
 R_c &= \{ \textit{this.fifo.head} && , \\
 & \quad \textit{this.fifo.data} && , \\
 & \quad \textit{this.fifo.num_elements} && , \\
 & \quad \textit{this.fifo.data_size} && \} \\
 W_c &= \{ \textit{this.fifo.head} && , \\
 & \quad \textit{this.fifo.num_elements} && \}
 \end{aligned}$$

5.2.3 Construction du graphe de conflit

La construction du graphe de conflit se fait en deux étapes.

1. Premièrement, il faut résoudre les chemins d'accès des différents ensembles de lecture et écriture en leurs objets réels déjà créés lors de l'étape d'élaboration. Pour cela, il faut procéder comme suit :
 - (a) Initialement il faut faire correspondre chaque racine **this** du chemin d'accès à son objet : Chaque transaction a été déclarée par un module spécifique. Ainsi la racine **this** dans le chemin d'accès correspond à l'objet déclarant la transaction.

- (b) Ensuite faire correspondre chaque chemin d'accès des ensembles de lecture et écriture avec le chemin dans le graphe de relation, construit précédemment, en prenant comme racine du chemin l'objet correspondant à `this`. Le dernier élément du chemin correspond à l'objet lu ou écrit.

Par exemple, le fait que le membre `fifo` dans le graphe de relation de la Figure 5.4 pointe vers le même nœud, alors l'objet correspondant à l'élément `this.fifo.head` de R_p est le même que celui correspondant à l'élément `this.fifo.head` de R_c . Il en est de même pour les autres membres de la `fifo`.

2. Une fois les chemins d'accès ont été résolus aux objets réels lus et écrits, nous procédons à la construction du graphe de conflit. Pour chaque transaction déclarée correspond un nœud du graphe. Une arête existe entre deux transactions s'il existe au moins un objet qui est lu par l'une de ces transactions et écrit par l'autre.

La Figure 5.5 présente le graphe de conflit de l'exemple du producteur et consommateur. Le graphe de conflit de la Figure 5.5 paraît trivial vu le modèle simple du

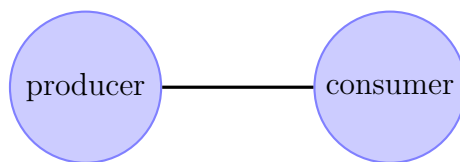


Figure 5.5 Graphe de Conflit de l'Exemple 5.2

producteur et consommateur choisi. Plus tard, dans le chapitre 6, nous allons évaluer une étude de cas industrielle s'agissant d'un modèle transactionnel spécifiant un transmetteur selon la norme Wi-Fi 802.11a.

5.3 Ordonnement

Dans cette étape nous allons essayer, sans avoir à incorporer des constructions de synchronisation, de trouver un ordonnancement des transactions qui maximise le parallélisme entre les transactions.

Pour qu'une exécution concurrente d'un modèle transactionnel soit correcte, il faut que le graphe de sérialisabilité correspondant à cette exécution soit acyclique (revoir section 4.7). Une condition suffisante pour ne pas générer un circuit dans le graphe de sérialisabilité est que deux transactions conflictuelles (ayant une arête commune dans le graphe de conflit) ne doivent pas s'exécuter en parallèle. Autrement dit, à un moment donné, seulement des transactions non conflictuelles peuvent s'exécuter en parallèle. L'avantage de cette technique est d'éviter d'ajouter des mécanismes de synchronisation au sein des transactions dans le but de minimiser le surcout de synchronisation. Par conséquent le problème d'ordonnement se réduit à partitionner les transactions en plusieurs groupes où les transactions de chaque groupe sont non conflictuelles et peuvent être exécutées d'une manière concurrente. Une fois ces groupes ont été déterminés, l'ordonnement consiste à exécuter les transactions sur plusieurs phases consécutives où chaque phase correspond à un groupe. Durant chaque phase, les transactions du même groupe correspondant à cette phase sont exécutées en parallèle. La Figure 5.6 montre l'ordonnement à exécuter une fois les transactions ont été partitionnées en n groupes où chaque groupe i est formé de l'ensemble des transactions $\{T_{i_1} \dots T_{i_k}\}$. L'exécution selon l'ordonnement spécifié se répète jusqu'à ce qu'aucune transaction n'est active.

Le problème de partitionnement des transactions se résume au problème de coloration de graphe de conflit[17]. C'est-à-dire attribuer une couleur à chacun des nœuds de telle sorte que deux nœuds reliés par une arête soient de couleurs différentes.

Dans le cas où nous ne connaissons pas le temps d'exécution des transactions, nous allons supposer qu'elles ont toutes le même temps d'exécution. Dans ce cas afin de minimiser le temps d'exécution total il faut trouver un partitionnement des

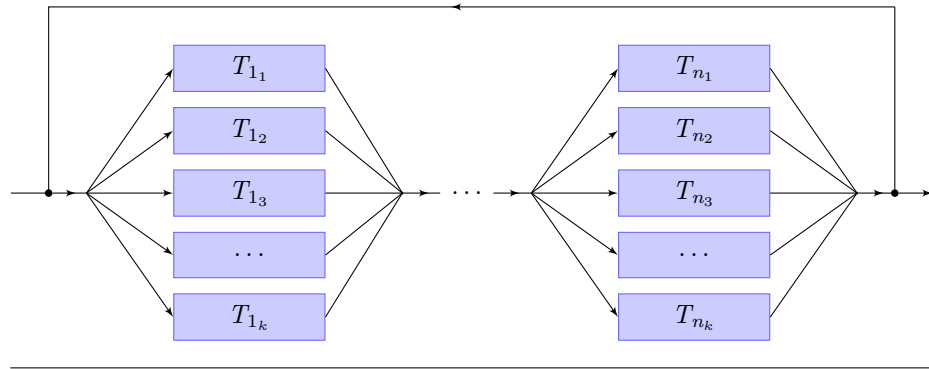


Figure 5.6 Ordonnement des transactions

transactions avec le minimum de groupes possibles. En d'autres termes colorier le graphe de conflit avec un nombre minimum de couleurs possibles, dit nombre chromatique.

La Figure 5.7 montre un exemple de coloration de graphe de conflit composé de 5 transactions. Dans ce cas le nombre chromatique est égal à 3 qui correspond aux 3 couleurs utilisées pour colorier les 5 nœuds du graphe.

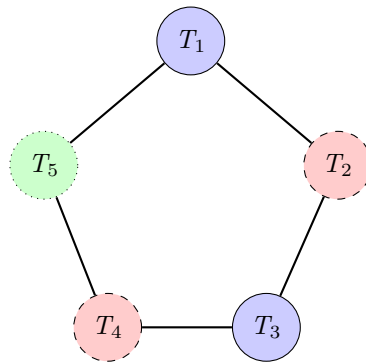


Figure 5.7 Exemple de coloration de graphe

Trouver un ordonnancement ayant un nombre de groupes égal au nombre chromatique serait suffisant au point de vue des performances si les transactions avaient approximativement la même charge de travail. Cependant ceci n'est pas toujours le cas. Ainsi lors du calcul de l'ordonnement il faut tenir compte du temps d'exécution des différentes transactions. Par exemple, si on prend l'or-

donnancement généré suite à la coloration du graphe de la Figure 5.7, le temps d'exécution, t_{iter} d'une itération serait égale à :

$$t_{iter} = \max(t_{T_1}, t_{T_3}) + \max(t_{T_2}, t_{T_4}) + t_{T_5}$$

Supposons maintenant que

$$t_{T_3} = t_{T_5} = x * t_{T_1} = x * t_{T_2} = x * t_{T_4} = x * t, x \geq 1$$

alors $t_{iter} = (2x + 1)t$. Cependant pour le même graphe et en tenant compte du temps d'exécution on peut retrouver une coloration (voir Figure 5.8) où l'ordonnancement correspondant aurait un temps d'exécution t'_{iter} égal à peu près à la moitié de t_{iter} :

$$t'_{iter} = \max(t_{T_1}, t_{T_4}) + \max(t_{T_3}, t_{T_5}) + t_{T_2} = (x + 2)t$$

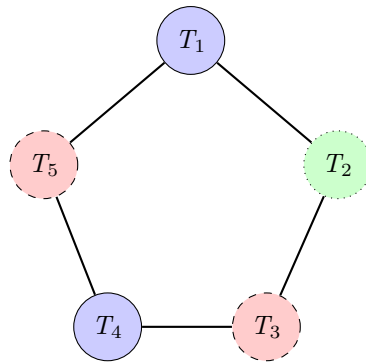


Figure 5.8 Nouvelle coloration du graphe de la Figure 5.7

Par conséquent dans le but d'avoir de meilleures performances de simulation nous avons tenu compte du temps d'exécution de chaque transaction. Pour estimer ce temps, nous profilons le modèle transactionnel pendant un nombre prédéfini d'itérations. À la fin de cette période de profilage, nous calculons pour chaque

transaction un temps d'exécution moyen en espérant que la période de profilage soit plus au moins fidèle à toute la période de simulation.

Une fois nous avons récupéré une estimation du temps d'exécution de chaque transaction, le problème d'ordonnancement se résume, à trouver le nombre optimal n de groupes de transactions qui minimise la fonction objective suivante :

$$\min \sum_{i=1}^n \max(t_{T_{i_1}}, \dots, t_{T_{i_k}}) \quad (5.1)$$

Ce problème est connu sous le nom de problème de coloration de graphes pondérés (« *Weighted Vertex Coloring Problem* » [45]). Nous avons résolu la fonction objective 5.1 à l'aide du solveur "IBM ILOG CPLEX Optimizer [33]". Nous avons utilisé en premier lieu une formulation qui paraît la plus naturelle et que nous décrivons comme suit [45].

Soit V l'ensemble de nœuds d'un graphe de conflit et soit E l'ensemble des arêtes. Nous attribuons à chaque nœud $i \in V$ un coût w_i qui est égal au temps d'exécution de la transaction correspondante. Notons que le nombre de couleurs (groupes) ne peut pas dépasser le nombre de nœuds n du graphe de conflit. Ainsi nous considérons une solution avec au maximum n couleurs. Par conséquent une manière directe afin de formuler le problème d'optimisation linéaire est de définir les deux ensembles suivants de variables :

- x_{ih} : Ce sont des variables binaires telles que $i \in V$ et $h = 1, \dots, n$. $x_{ih} = 1$ désigne le cas où le nœud i s'est fait attribuer la couleur h .
- z_h : Ce sont des variables réelles telles que $h = 1, \dots, n$. z_h désigne le coût de la couleur h . Il correspond au maximum des coûts des nœuds attribués à cette couleur h .

Voici la définition du problème d'optimisation linéaire afin de résoudre la fonction

objective de l'équation 5.1

$$\min \sum_{h=1}^n z_h \quad (5.2)$$

$$z_h \geq w_i x_{ih} \quad i \in V, h = 1 \dots n \quad (5.3)$$

$$\sum_{h=1}^n x_{ih} = 1 \quad i \in V \quad (5.4)$$

$$x_{ih} + x_{jh} \leq 1 \quad (i, j) \in E, h = 1 \dots n \quad (5.5)$$

$$x_{ih} \in \{0, 1\} \quad i \in V, h = 1 \dots n \quad (5.6)$$

La fonction objective 5.2 minimise la somme des coûts des couleurs z_h qui sont définies par les contraintes 5.3. Les contraintes 5.4 imposent que chaque nœud reçoive une couleur, et une seule, alors que les contraintes 5.5 exigent que deux nœuds adjacents ne reçoivent pas la même couleur. Finalement les contraintes 5.6 imposent aux variables x_{ih} d'avoir des valeurs binaires : soit 0 soit 1.

Le problème de partitionnement, comme décrit plus haut, ne spécifie pas une limite sur le nombre de transactions qui appartiennent au même groupe. Ainsi, pendant une phase d'ordonnement, le nombre de transactions peut dépasser le nombre de cœurs disponibles dans la machine de simulation. Dans ce cas, le parallélisme que nous avons obtenu va être réduit dû au nombre de cœurs limité. Plusieurs manières peuvent être envisagées pour ordonner les transactions d'une même phase. Les trois prochaines sous-sections vont décrire trois techniques différentes qui ont été implémentées puis évaluées dans le chapitre 6.

5.3.1 Ordonnement dynamique

L'ordonnement généré jusqu'à présent suppose que la machine de simulation possède un nombre suffisant de processeurs puisqu'aucune contrainte n'est imposée sur le nombre maximal que peut prendre un groupe de transactions non conflictuelles. Cependant lors de l'exécution concurrente des transactions il va y avoir au maximum autant de *threads* qui vont exécuter ces transactions que le nombre de cœurs présents dans la machine de simulation. Dans ce cas on parle du

concept de «*Thread Pool*» qui a été standardisé par l'introduction d'un patron de conception, «*Thread Pool pattern*»[58], décrivant son fonctionnement que voici.

Le principe de fonctionnement d'un «*Thread Pool*» est comme suit. Au lieu de créer un *thread* à chaque fois que nous avons besoin d'exécuter une tâche parallèle (dans notre cas il s'agit de transactions), nous allons créer un Pool de *threads* au début de lancement du programme principal. Lorsque ce dernier a besoin d'exécuter une tâche particulière d'une manière concurrente avec le reste de l'exécution, il l'enfile dans une file d'attente. Dès qu'un *thread* du pool devient disponible, il retire une tâche de la file d'attente puis l'exécute. Une fois le *thread* a fini d'exécuter une tâche, il la marque comme étant tâche complétée puis il retire une nouvelle tâche de la file d'attente. La Figure 5.9 illustre se comportement. Lorsqu'un *thread* a fini d'exécuter une tâche et la file d'attente est vide, il se met en mode inactif en attente d'une nouvelle tâche.

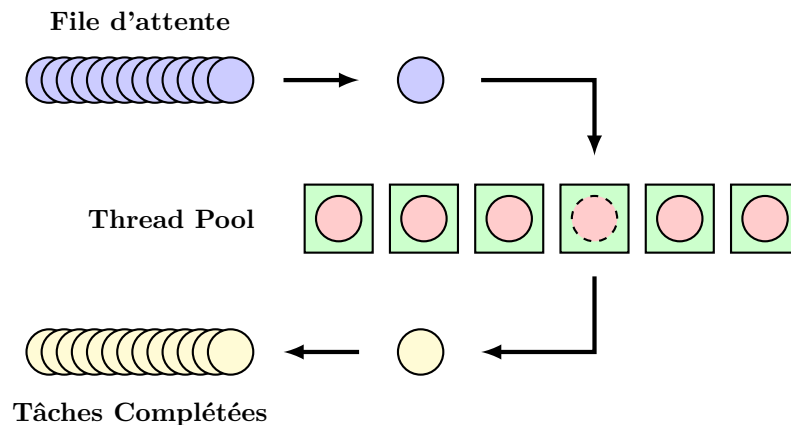


Figure 5.9 Un exemple de Thread Pool.

Il y a deux avantages majeurs liés aux performances lors de l'utilisation d'un pool de *threads* :

1. Avoir un *thread* prédisposé à exécuter des tâches lorsqu'elles se présentent comparé à créer un nouveau *thread* à chaque fois élimine le temps superflu nécessaire pour la création et la destruction de ce dernier. Créer et détruire un *thread* ainsi que ces ressources est une tâche coûteuse en terme de temps.

Ainsi l'utilisation d'un pool de *threads* permet d'avoir de meilleures performances.

2. Lorsque le nombre de tâches est assez important, comparé au nombre de cœurs présents, alors créer un *thread* pour chaque tâche va engendrer un nombre excessif de *threads* qui vont dégrader considérablement les performances. En effet, vu que le nombre de cœurs est petit par rapport au nombre de *threads* actifs alors il va y avoir un temps considérable gaspillé lors des changements de contexte entre ces derniers. Ainsi avoir un pool de *thread*, avec un nombre de *threads* ne dépassant pas le nombre de cœurs, élimine ce temps superflu dû aux changements de contexte.

5.3.2 Limiter le nombre de transactions

Une autre possibilité qui est intéressante à tester est de limiter le nombre de tâches parallèles à un moment donné aux nombres de cœurs présents dans la machine de simulation. Pour cela il faut que le nombre de transactions dans un groupe non conflictuel soit limité au nombre de cœurs. Ceci peut être spécifié en ajoutant une contrainte au problème d'optimisation linéaire qui limite le nombre de noeuds ayant la même couleur au nombre de cœurs N_c de la machine de simulation. Ceci est formulé à l'aide des contraintes 5.7 suivantes :

$$\sum_{i=1}^n x_{ih} \leq N_c \quad h = 1 \dots n \quad (5.7)$$

Malheureusement la résolution du problème d'optimisation, en se basant sur le modèle des équations 5.2-5.6 augmenté des contraintes de l'équation 5.7, est devenue impraticable (du moins pour les tests de la section 6.2) puisque le temps de résolution est devenu très lent et dépasse déjà le temps d'exécution d'une simulation séquentielle. Il est toujours possible de calculer l'ordonnancement une seule fois pour une architecture particulière du modèle, puis le réutiliser pour des simulations ultérieures. Toutefois, l'ordonnancement dépend aussi du temps d'exécution. Ainsi, si par exemple nous voulions mettre à jour l'ordonnancement au fur et à mesure que la simulation avance et le temps d'exécution de transactions

devient de plus en plus précis, alors nous avons intérêt à ce que la résolution du problème soit rapide. Dans ce cas-ci, nous pouvons utiliser une nouvelle formulation du problème qui soit plus efficace. En effet la formulation du problème utilisée contient certains inconvénients qui rend le temps de résolution très lent[45]. Un de ces inconvénients est que l'espace de solutions contient plusieurs solutions optimales à cause de la symétrie. En effet ayant trouvé une solution au problème, une autre solution existe en permutant tout simplement les couleurs entre les différents groupes. Ainsi pour réduire l'espace de solution il faut utiliser une nouvelle formulation qui supprime la symétrie. Dans ce qui suit nous décrivons cette nouvelle formulation qui a été proposée par [45] comme modèle *M2* à ce problème de partitionnement.

Dans la suite nous supposons que les nœuds du graphe de conflit i sont numérotés de telle manière que leurs coûts respectifs w_i suivent un ordre croissant, soit :

$$w_1 \geq w_2 \geq \dots \geq w_n.$$

Le modèle *M2* est basé sur la simple observation suivante. Pour chaque solution au problème de partitionnement il existe une solution équivalente où nous pouvons permuter les couleurs de telle façon que pour chaque groupe s tel que :

$$s = \{i_1, i_2, \dots, i_p\}$$

on lui réattribue la couleur i_1 qui correspond au premier nœud du groupe ayant le plus petit indice, ou en d'autres termes au nœud ayant le coût le plus important. Par conséquent pour ce modèle *M2*, nous considérons seulement les solutions de ce type. C'est à dire les solutions où chaque couleur h , si elle est utilisée (Il y a au moins un nœud qui lui a été attribué cette couleur), aurait un coût égal à w_h et peut être attribuée seulement aux nœuds $i \geq h$. Ainsi, nous introduisons une variable binaire y_{ih} prenant la valeur 1 si et seulement si le nœud i s'est fait attribué la couleur h ($i \in V, h \leq i$). Voici la définition du modèle *M2*.

$$\min \sum_{h=1}^n w_h y_{hh} \quad (5.8)$$

$$\sum_{h=1}^i y_{ih} = 1 \quad i = 1 \dots n \quad (5.9)$$

$$y_{ih} + y_{jh} \leq y_{hh} \quad (i, j) \in E, h = 1 \dots \min(i, j) \quad (5.10)$$

$$y_{ih} \leq y_{hh} \quad h = 1 \dots n, i > h \quad (5.11)$$

$$y_{ih} \in \{0, 1\} \quad h = 1 \dots n, i \geq h \quad (5.12)$$

$$\sum_{i=h}^n y_{ih} \leq N_c \quad h = 1 \dots n \quad (5.13)$$

La fonction objective 5.8 minimise la somme des coûts des couleurs utilisées. Les contraintes 5.9 et 5.10 sont les contreparties des contraintes 5.4 et 5.5 respectivement. Les contraintes 5.11 imposent à un nœud i de recevoir une couleur $h < i$ seulement si celle-ci a été déjà utilisée (déjà attribuée au nœud h). Les contraintes 5.12 imposent aux variables y_{ih} d'avoir les valeurs binaires $\{0, 1\}$. Finalement les contraintes 5.13 limitent le nombre des nœuds ayant la même couleur à N_c qui correspond au nombre de cœurs de la machine de simulation.

Avec ce nouveau modèle M2, nous avons pu passer d'un temps de résolution égale à des dizaines de secondes à un temps de résolution aux environs de 1/10 de secondes.

Puisque pendant un moment donné il ne peut y avoir plus que N_c tâches parallèles, nous pouvons, lors de l'exécution de l'ordonnancement, ne plus utiliser la file d'attente pour gérer les tâches, mais plutôt activer chaque *thread* séparément en lui spécifiant la tâche à exécuter. La Figure 5.10 illustre ce comportement. Ceci pourrait avoir une meilleure performance puisque le temps nécessaire à la synchronisation de la file d'attente entre les différents *threads* est éliminé.

5.3.3 Ordonnancement statique

Dans cette implémentation, comme c'était le cas de l'ordonnancement de la section 5.3.1, le problème utilisé pour générer les groupes de transactions ne spé-

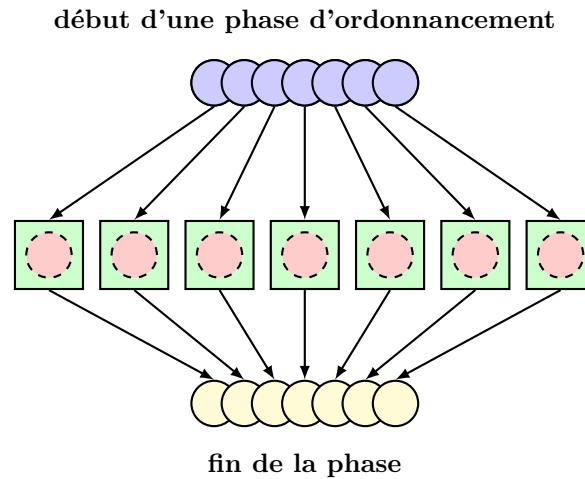


Figure 5.10 Thread Pool avec nombre de tâches ne dépassant pas le nombre de *threads*

cifie pas une limite sur le nombre de transactions par groupe. Cependant, au lieu d'exécuter les transactions dynamiquement en utilisant un pool de *threads* avec file d'attente, nous allons calculer un ordonnancement statique de telle façon que le nombre de tâches parallèles à exécuter durant chaque phase ne dépasse pas le nombre de cœurs disponibles. Par conséquent, au lieu d'exécuter une transaction par *thread* nous allons exécuter une séquence de transactions par *thread*. De cette façon nous espérons éliminer le surcoût dû à la synchronisation de la file d'attente.

Ce problème d'ordonnancement est bien connu sous le nom de «*minimum makespan scheduling*»[61] (ou «*multiprocessor scheduling*») qui se définit dans notre cas comme suit : Trouver une affectation des transactions aux différents *threads*, sachant que le nombre de *threads* ne dépasse pas le nombre de cœurs, de telle façon que le temps nécessaire pour finir l'exécution de toutes les transactions est minimisé. Comme précédemment, nous avons utilisé le solveur CPLEX[33] pour résoudre ce problème pour chaque phase d'ordonnancement (notée $phase_k$). Le problème d'optimisation linéaire est spécifié par les équations 5.14 à 5.16.

$$\min y \quad (5.14)$$

$$\sum_{j=1}^{N_c} x_{ij} = 1 \quad i \in phase_k \quad (5.15)$$

$$y \geq \sum_{i \in phase_k} w_i x_{ij} \quad j = 1 \dots N_c \quad (5.16)$$

La variable x_{ij} est égale à un si la transaction i est affectée au *thread* j autrement elle vaut zéro. Les contraintes 5.15 imposent à chaque transaction d'être affectée à un et un seul *thread*. La variable y est le temps d'exécution total à minimiser. Il doit être supérieur au délai d'exécution de chaque *thread* qui est égal à la somme des temps d'exécution des transactions affectées à ce *thread*, puisque dans un même *thread* les transactions s'exécutent séquentiellement. Ceci est imposé par les contraintes 5.16

5.4 Mise en œuvre en utilisant .NET

Dans cette section, nous présentons comment le modèle transactionnel est exécuté. Nous mettons l'accent sur les possibilités offertes par la plateforme .NET du point de vue de la réflexion et de la génération dynamique de code durant l'exécution.

À ce stade de la compilation nous avons un ordonnancement défini en tant qu'un tableau `slots` (voir listing 5.1). Chaque élément de ce tableau contient un ensemble de transactions. Le code du listing 5.1 montre comment nous procédons pour l'exécution de cet ordonnancement. Afin de garder le code clair, nous avons gardé seulement les instructions qui touchent à l'exécution de l'ordonnancement. Par exemple nous avons enlevé les instructions qui collectent les statistiques.

Listing 5.1 Code de l'ordonnanceur

```
while (true){
    // Vérifier si la période de profilage est finie.
    // Si c'est le cas déterminer un nouveau ordonnancement
    // basé sur les temps d'exécutions collectés.
```

```

if (newSchedule) slots = getScheduleByLP(true);

HashSet<TransObject> set = slots[slotNumber];

// S'il s'agit d'une seule transaction,
// elle est exécutée dans le thread courant,
// autrement les transactions sont encapsulées
// dans des tâches et exécutées dans le Thread Pool
if (set.Count == 1){
    genericTransaction method = set.ElementAt(0).action;
    sequentialResult = transaction(method);
}else {
    tasks = new Task<TaskResult>[set.Count];
    for (int i = 0; i < set.Count; i++)
    {
        genericTransaction method = set.ElementAt(i).action;
        tasks[i] = taskFactory.StartNew<TaskResult>(transaction, method);
    }
    Task.WaitAll(tasks);
}
// Si toutes les transactions ont été exécutées
// et aucune n'était active alors l'exécution est terminée
if (end) break;

// passer à la prochaine phase
slotNumber = (slotNumber + 1) % nbSlots;
}

```

La méthode `transaction` s'occupe de la gestion d'une transaction. Elle prend comme paramètre un objet de type `genericTransaction` qui représente la fonction qui s'est fait attribuer l'attribut `Transaction` dans le modèle transactionnel de départ. Nous allons voir comment créer un argument de ce type plus loin. Le listing 5.2 présente le code de la méthode `transaction`.

Listing 5.2 Encapsulation de la méthode avec l'attribut `Transaction` dans une méthode `transaction`

```

private TaskResult transaction(genericTransaction method){

```

```
//objet pour mesurer le temps d'exécution
ExecutionStopwatch elapsedTime = new ExecutionStopwatch();
elapsedTime.Start();

// Résultat de l'exécution
TaskResult result = new TaskResult();
try{
    Updater upd = new Updater();
    BaseModule.updater = upd;

    // Exécution de la transaction
    method();

    // Mise à jour des nouvelles valeurs des registres
    upd.update();

    result.Retry = false;
    result.duration = elapsedTime.Elapsed;
}catch (RetryException){
    result.Retry = true;
    result.durationRetry = elapsedTime.Elapsed;
}
return result;
}
```

Le type `genericTransaction` du paramètre `method` du listing 5.2 est un delegate. Un delegate dans .NET est une forme de type de fonction ou de pointeur de fonction. Ainsi une variable de type `genericTransaction` contient comme valeur une fonction. Cependant, puisqu'une fonction est associée généralement à une classe d'objets, lors de la création d'une variable de type delegate il faut spécifier aussi l'instance de cette classe. Ainsi, lors de l'appel de delegate, la fonction est appelée dans le contexte de cette instance. Le listing 5.3 montre comment nous créons une variable de type `genericTransaction` sachant que nous connaissons la fonction qui a été identifiée par l'attribut `Transaction` et l'instance d'objet qui correspond au module déclarant cette méthode. Cette instance est celle créée lors de la phase d'élaboration (voir sous-section 5.2.1)

Listing 5.3 Création d'un delegate genericTransaction

```
private genericTransaction Create(TransObject tranObject) {

    // type du module parent ayant défini la transaction
    Type type = tranObject.ParentObject.Instance.GetType();

    // Objet de réflexion fournissant les informations sur les méthodes
    MethodInfo method =
        type.GetMethod(tranObject.FuncSymbol.Name.ToString());

    // Création d'un delegate à partir de MethodInfo et
    // de l'objet parent(Instance) de la méthode
    genericTransaction transaction =
        (genericTransaction)Delegate.CreateDelegate(
            typeof(genericTransaction),
            tranObject.ParentObject.Instance,
            method);

    return transaction;
}
```

Au fait nous aurons pu utiliser la méthode classique afin d'appeler une méthode dynamiquement en liaison tardive. Il s'agit de l'appel `Invoke` de la classe `MethodInfo`. Cependant ceci est de l'ordre de 400 fois plus lent que l'utilisation de delegate. Et puisque chaque transaction est appelée plusieurs fois au cours de la simulation, l'utilisation du delegate va nous faire gagner considérablement au niveau du temps d'exécution comparé à l'utilisation classique de `Invoke`.

L'exécution de la transaction est initiée grâce à l'appel `taskFactory.StartNew<TaskResult>(transaction, method)`. Ceci va créer une tâche de type `Task<TaskResult>` puis appeler la fonction `QueueTask` de l'ordonnanceur de tâche `LockFreeTaskScheduler` qui a été spécifié grâce au code suivant :

```
taskScheduler = new LockFreeTaskScheduler();
taskFactory = new TaskFactory(taskScheduler);
```

La classe `LockFreeTaskScheduler` est une d'implémentation d'un « *Thread Pool* »

que nous avons déjà décrit à la section 5.3. Lors de la création de l'objet `task-Scheduler` Le pool de *threads* va être créé et les *threads* vont être lancés. Ceci est fait grâce au code suivant :

```
threadCount = Environment.ProcessorCount;

_threads = new Thread[threadCount];

for (int i = 0; i < threadCount; i++) {
    _threads[i] = new Thread(DispatchLoop) {
        Priority = threadPriority,
        IsBackground = true,
    };
    _threads[i].Start();
}
```

Remarquer que le nombre de *threads* créés est égal au nombre de cœurs logiques (`Environment.ProcessorCount`) présents sur la machine de simulation. Chaque *thread* va exécuter la fonction `DispatchLoop` définie comme suit :

```
private void DispatchLoop() {
    while (true) {
        Task wi;

        m_queue.Dequeue(out wi);

        while (wi == null) {
            // file d'attente vide.
            // attendre une nouvelle tâche
            new_task.WaitOne();

            // Si c'est un signal de fin terminer le thread.
            if (m_shutdown) return;

            m_queue.Dequeue(out wi);
        }
        // exécuter la tâche : transaction(method)
        TryExecuteTask(wi);
    }
}
```

```

    }
}

```

L'écriture dans la file d'attente est faite grâce à la fonction `QueueTask`. Comme il a été déjà mentionné, cet appel est initié par l'instruction `taskFactory.StartNew<TaskResult>(transaction, method)`. Voici le code de la méthode `QueueTask`.

```

protected override void QueueTask(Task task) {
    m_queue.Enqueue(task);
    // réveiller un thread pour exécuter la nouvelle tâche.
    new_task.Set();
}

```

Puisque plusieurs *threads* peuvent accéder à la file d'attente simultanément il faut la synchroniser afin d'éviter les erreurs dues aux concurrences critiques («*Race Condition*»). Les constructions utilisées pour la synchronisation auront une influence sur les performances. Pour cela nous avons codé deux implémentations :

- **bloquante** : Cette implémentation utilise des verrous pour protéger la file d'attente contre des accès concurrents
- **non bloquante** Dans la deuxième implémentation, nous utilisons des constructions de synchronisation non bloquantes. Voici un exemple de code d'une écriture non bloquante dans un segment de la file d'attente :

```

public bool TryEnqueue(Task node) {
    int head;
    int tail;
    GetIndexes(out head, out tail);
    do {
        if (tail == max_elements) {
            return false;
        }
    }
    // répéter si les valeurs de tail et head ont changé depuis
    // le moment où on a initié la lecture
    // avec GetIndexes(out head, out tail)
    while (!CompareExchangeIndexes(

```

```
    ref tail, tail+ 1, ref head, head));  
    // à ce stade nous avons réservé l'emplacement tail  
    nodes[tail] = node;  
    return true;  
}
```

Les résultats de ces deux implémentations ont été présentés dans la section 6.2.2.

Expérimentation

Dans ce chapitre nous allons présenter les résultats des expérimentations qui ont été effectuées pour évaluer la méthodologie proposée et mesurer les performances de simulations obtenues suite à l'application des phases de compilation présentées plus haut.

En premier lieu nous présentons un aperçu sur l'étude de cas utilisée pour l'expérimentation. Il s'agit d'un modèle de transmetteur selon la norme Wi-Fi 802.11a. Ensuite nous décrivons les différents tests effectués tout en donnant et analysant les résultats obtenus.

6.1 Modèle du transmetteur 802.11a

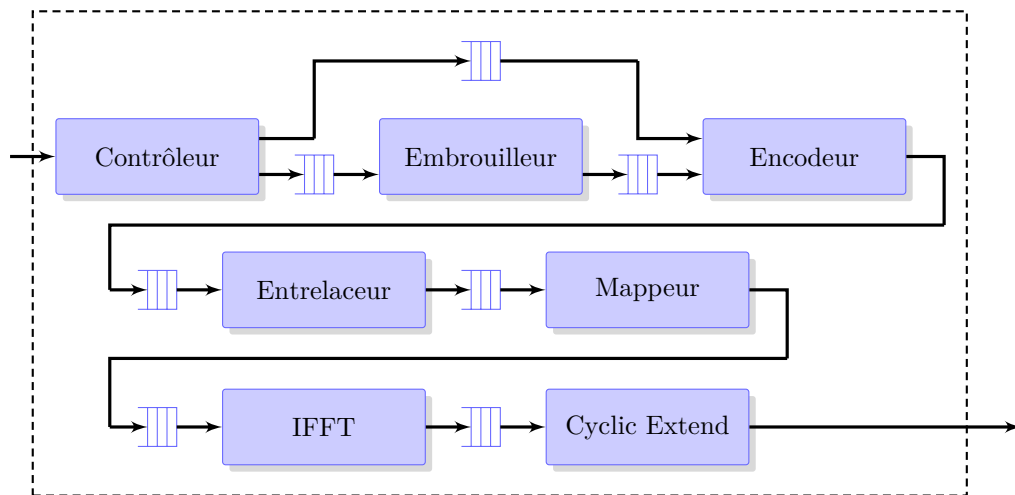


Figure 6.1 Modèle du transmetteur 802.11a.

L'étude de cas que nous allons présenter dans cette section est un transmetteur de données par liaison sans fil selon la norme IEEE 802.11a (Wi-Fi)[57]. C'est une

adaptation de l'étude de cas[18] faite en Bluespec vers le modèle transactionnel défini dans l'environnement Microsoft .Net. La Figure 6.1 montre les modules qui composent le modèle du transmetteur 802.11a.

6.1.1 Contrôleur

Le contrôleur reçoit les paquets de la couche MAC comme un flux de données. Ensuite il est responsable de créer l'entête de chaque paquet à transmettre et de faire en sorte que la partie de données de chaque paquet contient les informations de contrôle nécessaires telles que le débit de transmission à utiliser. Une fois prête, l'entête est envoyée directement vers l'encodeur alors que les données ont besoin d'être embrouillées et elles sont envoyées à l'embrouilleur.

Les paquets sont traités par le contrôleur grâce à deux transactions "sendHeader" et "sendData". La transaction "sendHeader" lit les informations d'un paquet à partir d'un fichier de test, appelle la fonction du contrôleur pour la gestion de l'entête, puis passe la main à la transaction "sendData". Cette dernière appelle la fonction de contrôleur pour la gestion des données. Les données sont traitées par unité de taille n prédéfinie(24 bits). Ainsi cette transaction s'exécute plusieurs fois jusqu'à ce que toutes les données aient été traitées, puis elle repasse la main à la transaction "sendHeader" pour traiter le prochain paquet.

6.1.2 Embrouilleur

L'embrouilleur fait un ou exclusif (xor) sur les données de chaque paquet en utilisant un motif pseudo-aléatoire de bits. Le motif utilise un registre à décalage de 7 bits et deux portes XORs. La Figure 6.2 montre le bit de sortie embrouillé généré pour chaque bit de donnée en entrée. Toute la fonction d'embrouillage est exécutée par une la transaction "Scramble" qui lit les données de la FIFO d'entrée puis génère les données embrouillées dans la FIFO de sortie.

6.1.3 Encodeur convolutionnel

L'encodeur contient deux transactions "sort" et "encode". La transaction "sort" permet de détecter s'il s'agit d'un nouveau paquet. Dans ce cas, elle transfère l'entête du nouveau paquet de la FIFO d'entrée vers la FIFO de sortie avant

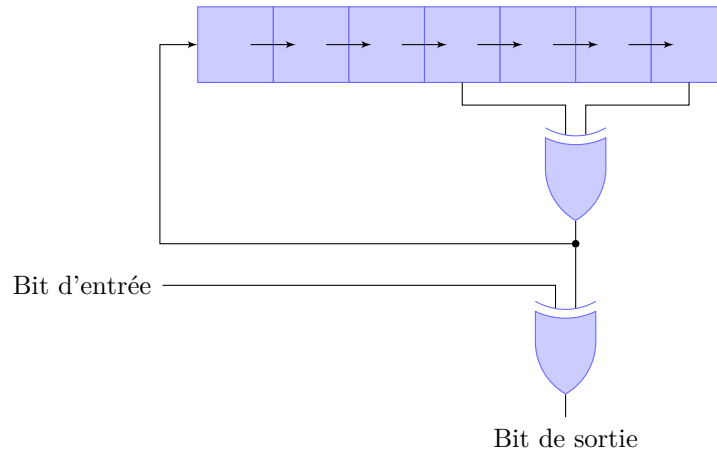


Figure 6.2 Fonction d'embrouillage

de commencer à traiter les données. La deuxième transaction “encode” permet d’implémenter la fonction d’encodage décrite dans la Figure 6.3. Pour chaque bit d’entrée, deux bits de sortie sont générés en utilisant un registre à décalage et trois fonctions XORs.

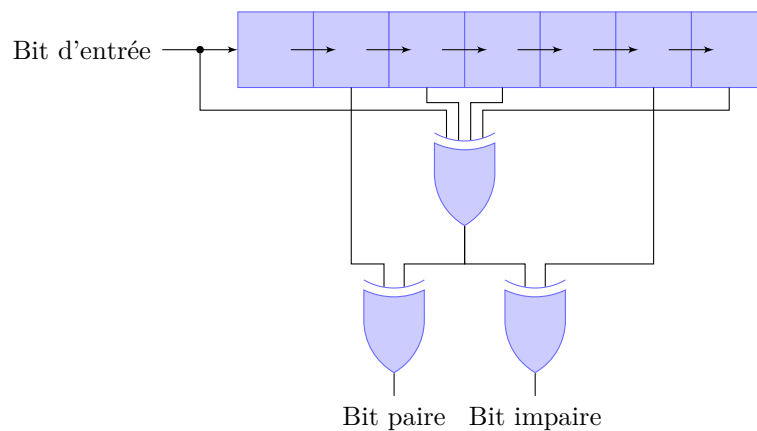


Figure 6.3 Fonction d’encodage

6.1.4 Entrelaceur

L’entrelaceur fonctionne au niveau du symbole OFDM avec des tailles de bloc, N_{cbps} de 48, 96 ou 192 bits selon le débit utilisé. Pour chaque bloc, les bits encodés

sont permutés en deux étapes. Dans la première étape, ils sont réordonnés sur des sous-porteuses non adjacentes. Pendant la deuxième étape, les bits originellement adjacents de la même sous-porteuse sont permutés alternativement vers les bits de poids fort et faible de la même sous-porteuse.

Soit k l'indice du bit encodé avant entrelacement, i l'indice après la première étape, j l'indice après la deuxième étape et $s = \max(N_{cbps}/2, 1)$.

La première permutation i est définie par :

$$i = \left(\frac{N_{cbps}}{16} \right) (k \bmod 16) + \text{floor} \left(\frac{k}{16} \right) \quad k = 0, 1, \dots, N_{cbps} - 1$$

La deuxième permutation j est définie par :

$$j = s \times \text{floor} \left(\frac{i}{s} \right) + \left(i + N_{cbps} - \text{floor} \left(16 \times \frac{i}{N_{cbps}} \right) \right) \bmod s,$$

où $i = 0, 1, \dots, N_{cbps} - 1$

Étant donné que ces indices peuvent être calculés à l'avance, l'entrelaceur réorganise essentiellement les bits d'entrée selon un ordre précalculé dépendant du débit de données. Toute la fonction d'entrelacement est implémentée par une la transaction "interleaver".

6.1.5 Mappeur

Le mappeur opère également au niveau symbole OFDM avec une taille de bloc de 48, 96 ou 192 bits. Chaque bloc est divisé en sous-blocs au niveau sous-porteuse OFDM. Les sous-blocs sont de taille 1, 2 ou 4 bits. Le mappeur tout d'abord convertit chaque sous-bloc en un nombre complexe représentant les points de constellation de BPSK, QPSK, ou 16-QAM selon le débit utilisé pour le paquet en cours. Notez que le type de modulation peut être différent pour les parties d'en-tête et les données du message puisque les entêtes sont toujours transmises en BPSK. Les 48 nombres complexes résultants sont ensuite normalisés par un facteur de normalisation dépendant du débit puis rembourrés avec des pilotes

et des zéros pour finir avec 64 nombres complexes. Toute cette fonctionnalité est implémentée à l'intérieur de la transaction “map”.

Avant que les 64 nombres complexes ne soient envoyés à la IFFT pour transformation, ils sont réordonnés selon leur indice de fréquence dans le symbole OFDM. Cette permutation est exécuté à l'aide de la transaction “reorder”.

6.1.6 IFFT

La IFFT contient une transaction “ifft” permettant d'effectuer la transformée de Fourier rapide inverse sur les 64 valeurs de fréquence complexes pour les traduire dans le domaine temporel où ils peuvent ensuite être transmises sans fil.

6.1.7 Extenseur de cycle

Après la transformée de Fourier, l'extenseur de cycle prolonge le symbole IFFT généré en ajoutant le début du message au corps du message complet formant ainsi 81 nombres complexes représentant le signal à transmettre pour le symbole OFDM en cours. Ceci est implémenté grâce à la transaction “cyclicExtender”.

6.2 Résultats expérimentaux

Nous allons commencer par décrire le cadre expérimental dans lequel se sont déroulés les tests. Ensuite nous détaillons les expériences réalisées ainsi que les résultats obtenus.

6.2.1 Cadre expérimental

Les tests de performance consistent à mesurer le temps d'exécution de plusieurs configurations en variant à chaque fois le nombre de *threads* parallèles utilisés pendant la simulation. Ce temps d'exécution correspond au temps écoulé entre le moment du lancement de l'application et la fin de l'exécution. Cependant ce temps peut varier considérablement entre une exécution et une autre. Cette variation est due au fait que le temps mesuré est le temps pendant lequel l'application est exécutée par le processeur en plus du temps inactif pendant lequel l'application était en attente. Puisque ce temps d'attente change d'une exécution à une autre,

le temps d'exécution total varie lui aussi. En effet le temps d'attente peut être dû principalement aux raisons suivantes :

- Un accès au disque dur.
- L'exécution du Ramasse-miettes (« *Garbage Collector* »).
- Défaut de page.
- Exécution d'un autre *thread*.

Les trois premières raisons dépendent essentiellement de la charge de travail de l'application en cours et le temps d'attente dû à ces raisons varie d'une façon négligeable entre une exécution et une autre. Cependant lorsque plusieurs *threads* sollicitent un même processeur, le système d'exploitation distribue le temps du processeur sur ces différents *threads* et oblige ainsi chaque *thread* d'attendre son tour. Ainsi le temps d'attente est proportionnel aux nombres de *threads* actifs prêts à être exécutés. Ce nombre varie d'un moment à un autre selon la charge de travail du système d'exploitation et celle de toutes les applications en cours d'exécution.

Par conséquent afin de mesurer le temps d'exécution écoulé nous pouvons procéder de plusieurs façons différentes :

- Arrêter les services qui ne sont pas nécessaires au fonctionnement de la machine afin de minimiser au maximum l'interaction avec le modèle sous test. Ensuite, faire plusieurs exécutions du même test puis prendre la moyenne du temps d'exécution écoulé comme mesure.
- Mesurer le temps pendant lequel l'application était en attente des processeurs pendant qu'ils servaient d'autres *threads* appartenant à d'autres processus, puis soustraire ce temps du temps total mesuré. Ce type de profilage nécessite l'interception de tous les changements de contextes puis analyser la raison de chaque changement de contexte et la durée de chaque attente pour ne tenir que ceux qui sont liés à l'exécution de l'application. Ce genre

de profilage est rendu possible dans une machine de la famille «*Windows*» grâce à la génération des événements («*Event Tracing for Windows*»).

- Approximer une situation idéale où l'application s'exécute dans un environnement fermé sans interaction avec les autres processus. Pour cela nous pouvons augmenter la priorité du processus de l'application en cours de test. Ceci va permettre de privilégier l'exécution de l'application en cours sans être interrompue par les autres processus, et ainsi avoir des mesures avec des marges d'erreur moins importantes.

Nous avons opté pour la combinaison de la première et la troisième option pour calculer le temps d'exécution. En effet, en augmentant la priorité du processus de l'application sous test ainsi que tous ces *threads* et en gardant une charge de travail faible et constante du système d'exploitation nous avons pu avoir des temps d'exécutions dont la variation est beaucoup moins importante. En plus, afin de minimiser encore les perturbations dans les résultats nous avons fait une moyenne des temps d'exécutions après avoir répété les tests une dizaine de fois pour chaque expérience.

Les différents tests ont été faits en utilisant un fichier de test comportant cent paquets. La taille et les données de chaque paquet ont été générées aléatoirement. En plus des transactions décrites plus haut, une autre transaction, "result", est ajoutée pour extraire les résultats calculés. Les données de sortie obtenues sont identiques comparées aux résultats générés par le modèle initial de Bluespec.

6.2.2 Résultats expérimentaux

Le modèle du Transmetteur 802.11a contient en tout 11 transactions. Après résolution du problème de coloration de sommets sur le graphe de conflit selon les contraintes (5.2-5.7) on obtient le graphe colorié de La Figure 6.4. En tout, on obtient deux groupes de transactions où dans chaque groupe les transactions sont non conflictuelles et peuvent être exécutées en parallèle.

La Figure 6.5 montre l'ordonnancement des transactions utilisé pendant l'exécution parallèle suite à la coloration obtenue précédemment. L'ordonnancement

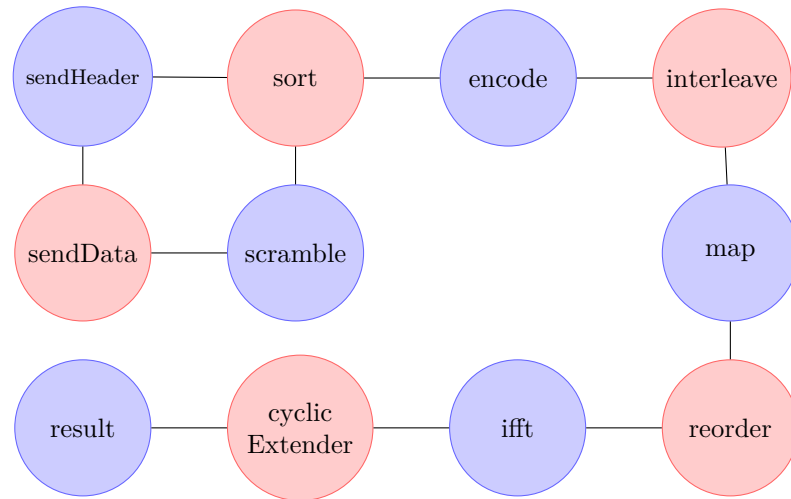


Figure 6.4 Coloration du Graphe de Conflit

se fait en deux phases. Pendant la première phase, les transactions du premier groupe sont exécutées par le pool de *threads* en passant par une file d'attente comme décrit dans la section 5.3.1. Lorsque toutes les transactions ont terminé leurs exécutions, l'ordonnanceur passe à la deuxième phase et exécute de la même manière les transactions du deuxième groupe. Le cycle se répète jusqu'à ce que toutes les transactions deviennent inactives.

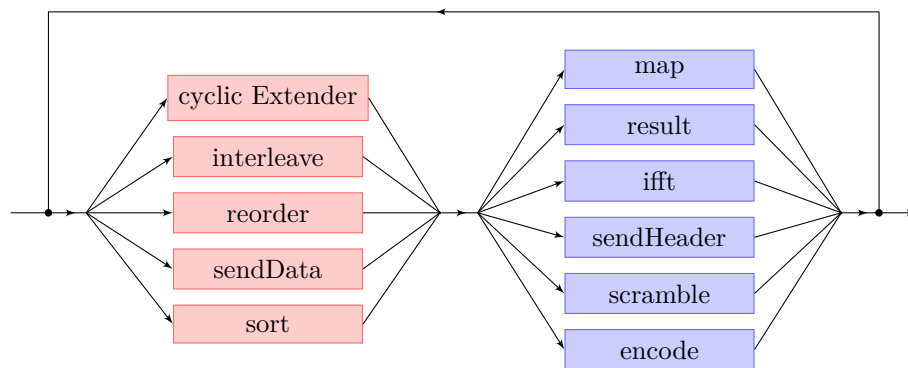


Figure 6.5 Exécution Parallèle

Le graphe de la Figure 6.6 montre le temps d'exécution de l'étude de cas selon l'ordonnancement de la Figure 6.5. Le cas où le nombre de *threads* est égal à 1

correspond à une exécution séquentielle. Pour les autres cas, le nombre de *threads* correspond aux nombres de *threads* créés dans le «*Thread Pool*».

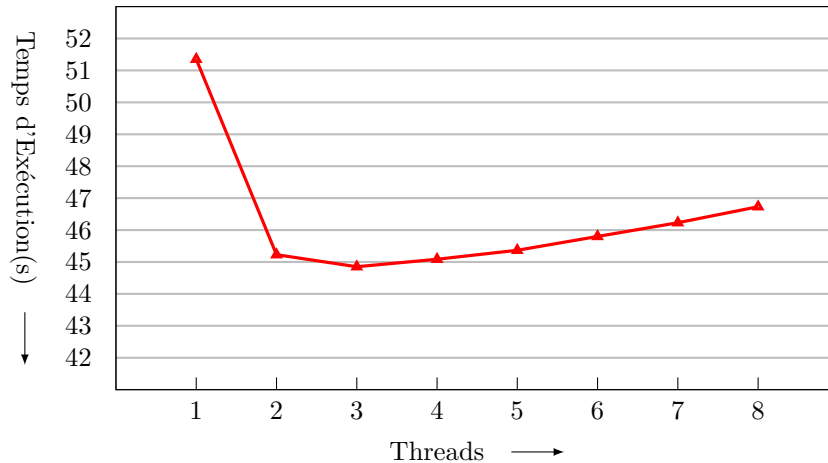


Figure 6.6 Temps d'exécution suite à l'ordonnancement de la Figure 6.5

Nous remarquons que lorsque le nombre de *threads* est égal à 3 nous obtenons la meilleure accélération qui est égale à $\frac{51.35}{44.85} \approx 1.14$. À ce stade deux questions se posent :

- Pourquoi obtenons-nous une faible accélération ?
- Pourquoi le fait d'avoir quatre *threads* ou plus n'améliore-t-il pas les performances, si ce n'est pas le contraire ?

Afin de répondre à ces deux questions, il faut consulter les temps d'exécutions des différentes transactions. Le Tableau 6.1 montre ce temps d'exécution pour chaque transaction calculé lors d'une exécution séquentielle de la présente étude de cas ainsi que le pourcentage du temps d'exécution total.

Ainsi la faible accélération est due au fait que 81% du traitement est consacré à la transaction `ifft`. Cela nous laisse au plus 20% de traitement à paralléliser. Pour illustrer mieux ceci, essayons de comparer l'accélération obtenue par rapport à l'accélération maximale que nous pouvons atteindre. Pour cela nous

Tableau 6.1 Temps d'exécution

	Temps d'Exécution(s)	% Temps Total
sendHeader	0,68	1,35%
sendData	0,08	0,16%
result	3,57	7,09%
scramble	0,55	1,09%
interleave	0,92	1,83%
map	1,08	2,14%
reorder	1,39	2,76%
cyclicExtender	0,51	1,02%
sort	0,04	0,09%
encode	0,60	1,19%
ifft	40,93	81,23%

allons appliquer la loi d'Amdahl. Soit P la portion du programme qui peut être parallélisée. Ainsi l'accélération maximale que nous pouvons avoir, quelque soit l'ordonnancement utilisé et étant donné N processeurs, est :

$$\frac{1}{(1 - P) + \frac{P}{N}}$$

Puisque 81% du temps d'exécution est consommé par la transaction `ifft`, la portion du programme que nous pouvons paralléliser est au plus égale à $P = 0.19$. Ce qui nous donne étant donné 8 processeurs une accélération maximale de :

$$A_{max} = \frac{1}{(1 - 0.19) + \frac{0.19}{8}} \approx 1.2.$$

Elle est toujours aussi faible que l'accélération de 1.14 que nous avons obtenu. Ceci répond à la première question.

Maintenant pour répondre à la deuxième question nous avons comparé le résultat obtenu par rapport au temps d'exécution d'une simulation hypothétique

qui se déroule sans surcoûts. Afin d'estimer ce temps d'exécution nous procédons de la manière suivante.

1. Durant une simulation séquentielle, les transactions s'exécutent en isolation sans interférence avec d'autres *threads* de l'application puisqu'il en existe qu'un. Cependant lors d'une simulation concurrente plusieurs *threads* s'exécutent en parallèle. Cette interférence entre les *threads* pourrait engendrer des surcoûts au niveau des temps d'exécution. Et puisque nous voulons estimer une exécution idéale, nous nous basons sur le temps d'exécution de chaque transaction mesuré durant une exécution séquentielle pour en déduire le temps d'exécution totale.
2. Étant donné le temps d'exécution de chaque transaction durant une phase d'ordonnancement, l'algorithme présenté dans le Listing 6.1 estime le temps d'exécution nécessaire pour finir l'exécution de toute la phase. Si le nombre de transactions est inférieur aux nombres de *threads*, alors ce temps d'exécution sera le maximum entre les temps d'exécution des transactions de la même phase. Si le nombre de transactions est supérieur aux nombres de *threads*, l'algorithme va simuler une exécution dynamique des transactions dans un pool de *threads* en se basant sur les temps d'exécution de chaque transaction. Soit t_{ij} les temps d'exécution des transactions durant la phase j de l'itération i . Le temps d'exécution total est alors égal :

$$t_{tot} = \sum_{i=1}^n \sum_{j=1}^k estimateExecutionTime(t_{ij})$$

Pour pouvoir estimer ce temps t_{tot} il faut pouvoir calculer et sauvegarder le temps d'exécution séquentiel qu'aurait pu prendre chaque transaction durant chaque itération. Afin de simplifier le calcul, nous supposons que l'ordonnancement obtenu suite à l'exécution dynamique des transactions ne varie pas beaucoup entre les itérations. Ceci reste vrai pour l'étude de cas en cours. Ainsi nous pouvons simplifier le temps d'exécution total par


l'équation suivante :

$$t_{tot} = \sum_{j=1}^k estimateExecutionTime\left(\sum_{i=1}^n t_{ij}\right) \quad (6.1)$$

La somme $\sum_{i=1}^n t_{ij}$ correspond aux temps d'exécution total que prend l'exécution de chaque transaction durant toute la simulation de l'étude de cas.

Listing 6.1 Déterminer une estimation du temps d'exécution

```
double estimateExecutionTime(extime)
{
    // trans-list est la liste des transactions
    // ordonnée par temps d'exécution décroissant ;
    trans-list = liste ordonnée des transactions;
    // chaque élément i du tableau threads-sched
    // représente la séquence des transactions
    // que le thread i va exécuter
    pour tout thread i threads-sched[i] = {};
    // chaque élément i du tableau threads-time
    // représente le temps qui reste au thread i
    // pour terminer l'exécution de la transaction courante
    pour tout thread i threads-time[i] = 0;
    while ((|ready-list| > 0)) {
        while (il existe un thread i tel que threads-time[i] = 0
            && |ready-list| > 0) {
            tran = trans-list.dequeue();
            threads-sched[i].add(tran);
            threads-time[i] = extime[tran];
        }
        // trouver thread n tel que
        // threads-time[n] est minimum
        n = getNextThread(threads-time);
        pour tout thread i threads-time[i] -= threads-time[n];
    }
    // trouver le temps d'exécution
    // de la plus longue séquence de transactions
    double exeTime = getMaxSequence(threads-sched);
    return exeTime;
}
```

Le graphe, , de la la Figure 6.7 représente le meilleur temps d'exécution calculé selon l'équation 6.1 en variant à chaque fois le nombre de *threads* utilisés.

Nous remarquons qu'au-delà de trois *threads* le temps d'exécution que nous pouvons encore gagner est quasi nul. Ainsi le fait d'augmenter le nombre de *threads*

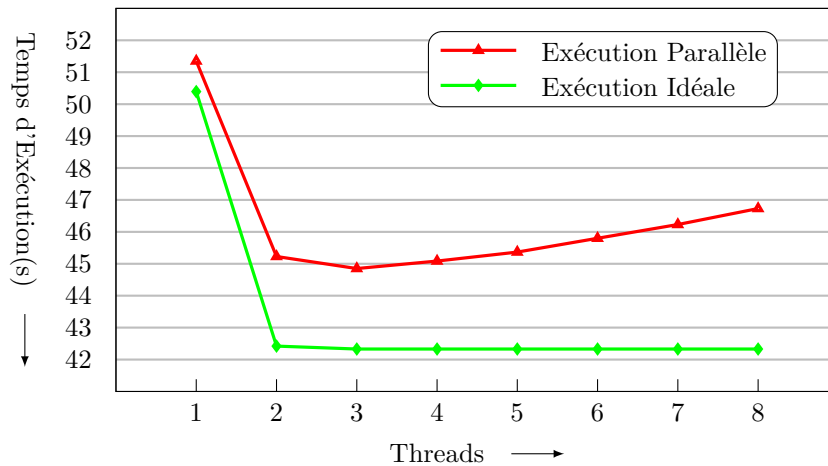


Figure 6.7 Comparaison des Temps d'exécution obtenus par rapport aux meilleurs temps que nous pouvons obtenir.

fait augmenter le surcoût sans pour autant améliorer le temps d'exécution d'où on commence à perdre au niveau des performances à partir de l'utilisation de quatre *threads*.

Ce premier test montre en quelque sorte les limitations du parallélisme que nous générons. En effet ce que nous essayons de faire est de partitionner l'ensemble des transactions en plusieurs groupes parallèles. Ainsi dans un cas extrême dans lequel un modèle est défini par une seule transaction, tout ce que nous pouvons faire est d'exécuter cette transaction dans un seul *thread* sans pouvoir profiter d'une machine de simulation multicœur (Il est possible toutefois de pouvoir décomposer une transaction en plusieurs transactions que nous envisageons de faire dans des travaux ultérieurs).

Cependant l'implémentation de la `ifft` utilisée dans ce premier test correspond à un circuit combinatoire qui lui aussi va engendrer un chemin critique très important lors de la synthèse matérielle du modèle transactionnel. Ainsi le concepteur est obligé de spécifier une nouvelle architecture de la `ifft` afin de minimiser le chemin critique et ainsi minimiser la fréquence de l'horloge. Dans cette nouvelle architecture, le concepteur va devoir décomposer la transaction `ifft` en plusieurs

autres transactions. Grâce à ceci il serait possible de pouvoir paralléliser une plus grande portion du modèle qui pourrait engendrer une meilleure accélération.

Une nouvelle architecture possible de la `ifft` est de la décomposer en trois étages pipelinés où chaque étage est traité par 4 unités d'exécutions parallèles. La Figure 6.8 représente les différentes transactions qui décrivent un des étages du pipeline. Tous les étages suivent la même décomposition.

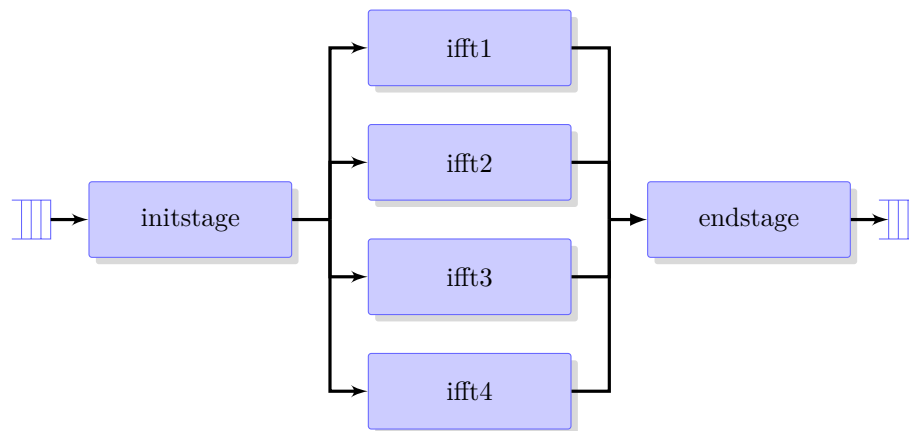




Figure 6.8 Un étage parmi les trois étages de la nouvelle IFFT Pipelinée.

Avec cette nouvelle spécification nous avons passé d'une seule transaction `ifft` qui décrivait toute la fonctionnalité de la IFFT à une architecture pipelinée et parallèle composée de $6 * 3 = 18$ transactions.

Le graphe  de la Figure 6.9 montre les temps d'exécutions de l'étude de cas en utilisant l'architecture pipelinée de la Figure 6.8 en augmentant à chaque fois le nombre de *threads* utilisés. Grâce à cette nouvelle architecture nous avons pu atteindre une accélération égale à **4.25**.

Le graphe  de la Figure 6.9 montre le temps d'exécution que nous pourrions avoir si nous supprimons tout le surcoût. Dans ce cas nous aurions pu atteindre une accélération de **6.60**. Tout le long des tests nous avons essayé plusieurs configurations dans le but de minimiser ce surcoût. Les résultats de la Figure 6.9 sont les meilleurs que nous avons pu obtenir. Dans la section qui suit, nous al-

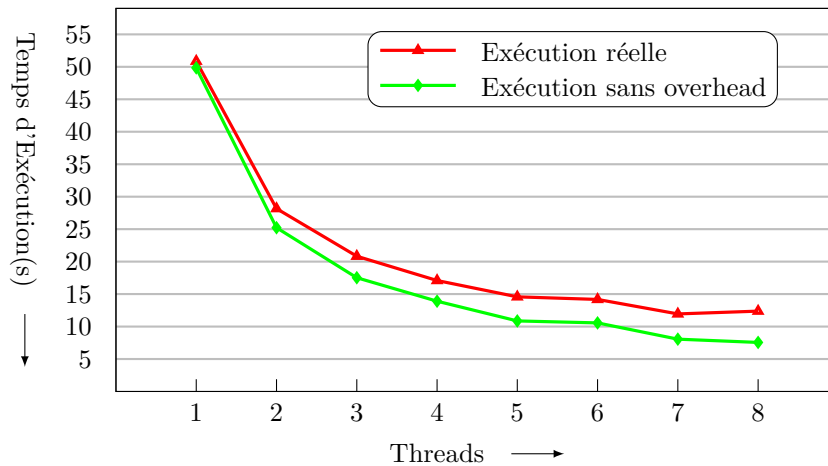


Figure 6.9 Temps d'exécution en utilisant l'architecture pipelinée de la Figure 6.8

lons présenter les différents implémentations et tests que nous avons essayés avant d'arriver à ces résultats.

6.2.3 Implémentations et tests

Afin de pouvoir minimiser le surcoût, il faut détecter les fonctionnalités additionnelles par rapport à une exécution séquentielle et essayer de diminuer leurs impacts sur le temps d'exécution. Voici les principales causes qui génèrent du surcoût :

- Le code d'instrumentation afin de collecter et calculer le temps d'exécution. Ce surcoût a été écarté le plus que possible des mesures effectuées afin qu'il n'influe pas sur les résultats.
- Le traitement supplémentaire lors de l'exécution de chaque phase d'ordonancement. Ce traitement peut être divisé en deux parties :
 - **surcoût dû à la gestion des phases** : Durant chaque phase d'ordonancement il faut encapsuler les transactions dans des tâches, réveiller les *threads* pour qu'ils lancent l'exécution des tâches, exécuter les transactions puis se synchroniser avec les *threads* afin de pouvoir passer à

la prochaine phase une fois toutes les tâches ont été exécutées. Ainsi à part l'exécution proprement dite de chaque transaction, le reste est compté comme surcoût par rapport à une exécution séquentielle.

- **surcoût dû à la synchronisation de la file d'attente** : C'est le temps nécessaire pour la synchronisation de la file d'attente.


Nous allons appeler le temps additionnel causé par toutes ces fonctionnalités **surcoût d'ordonnement**.

- Durant une simulation séquentielle, les transactions s'exécutent en isolation sans interférence avec d'autres *threads* de l'application puisqu'il en existe qu'un. Cependant lors d'une simulation concurrente plusieurs *threads* s'exécutent en parallèle. Ceci peut engendrer du surcoût pour les raisons suivantes :

- **changements de contexte** :Lorsqu'il existe plus de *threads* que de cœurs, le système d'exploitation doit les ordonner à travers plusieurs files de priorité. Ceci va générer du surcoût dû aux changements de contextes entre les différents *threads*. Puisque dans notre cas nous limitons le nombre de *threads* au maximum au nombre de cœurs de la machine de simulation le surcoût dû aux changements de contextes est négligeable et n'a pas presque d'influence sur les résultats.
- **ramasse-miettes(Garbage Collector)** :Le «*Garbage Collector*» est responsable de la libération de l'espace mémoire des objets qui ne sont plus référencés. Pour ce faire il doit se synchroniser avec tous les *threads* de l'application et les suspendre momentanément si nécessaire pour pouvoir exécuter une collection de nettoyage. Ceci a beaucoup d'influence sur l'exécution de la simulation et peut engendrer un surcoût important dû aux temps de synchronisation et d'attente.

Nous appelons le temps additionnel causé par le fait d'exécuter plusieurs *threads* concurrentement surcoût d'interférence.

Suite à cette discussion, nous considérons principalement deux types majeurs de surcoûts : surcoût d’ordonnancement et surcoût d’interférence. Afin de mesurer l’impact de ces deux types de surcoût, nous procédons de la manière suivante.

1. Nous utilisons l’équation 6.1 afin d’estimer le temps d’exécution d’une simulation hypothétique qui se déroule sans surcoûts et qui génère le meilleur temps d’exécution possible. Les graphes  de la Figure 6.9 et de la Figure 6.7 sont les résultats de ce calcul. Appelons ce temps d’exécution $t_{parfait}$.
2. Nous calculons le temps d’exécution total de chaque transaction durant l’exécution parallèle. Ce temps d’exécution et le temps écoulé depuis le début de l’exécution d’une transaction jusqu’à la fin. Ainsi il n’inclut pas le surcoût d’ordonnancement, mais seulement le surcoût d’interférence. Maintenant comme nous avons fait dans le premier cas, étant donné un ordonnancement des transactions et un nombre de *threads*, nous prenons ce temps d’exécution pour estimer le temps d’exécution d’une simulation hypothétique qui se déroule avec seulement le surcoût d’interférence en utilisant l’équation 6.1. Appelons ce temps d’exécution $t_{interference}$.
3. Maintenant si on soustrait le temps $t_{parfait}$ du temps d’exécution de la simulation parallèle t_{real} nous obtenons la perte totale due aux surcoûts. De même $t_{interference} - t_{parfait}$ donne la perte due au surcoût d’interférence et $t_{real} - t_{interference}$ donne la perte due au surcoût d’ordonnancement. La Figure 6.10 montrent l’impact des surcoûts durant les tests de la Figure 6.9.

Différentes implémentations ont été testées dans le but de minimiser les surcoûts d’ordonnancement. Voici quatre implémentations dont les résultats expérimentaux vont être présentés plus loin :

- **“Dynamique”** : Cette configuration utilise un pool de *threads* avec file d’attente. Pour plus d’information, se référer à la section 5.3.1.

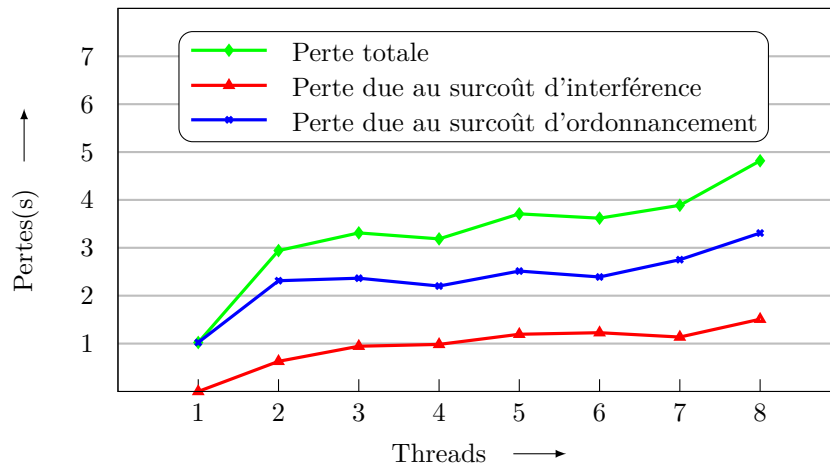


Figure 6.10 Pertes dues aux surcoûts.

- **“Dédiée”** : Cette configuration utilise un ordonnancement où dans chaque phase le nombre de transactions allouées ne dépasse pas le nombre de *threads* utilisés. Elle est déjà décrite dans la section 5.3.2.
- **“Statique”** : Cette configuration exécute durant chaque phase un ordonnancement statique. La manière dont nous calculons cet ordonnancement est déjà présentée dans la section. 5.3.3.
- **“Avec verrous”** : La file d’attente des transactions utilisée dans l’implémentation dynamique se base sur des constructions de synchronisation non bloquantes pour se protéger contre les accès concurrents de plusieurs *threads*(voir section 5.4). Dans cette implémentation nous remplaçons la synchronisation non bloquante de la file d’attente par une bloquante en utilisant des verrous.

La Figure 6.11 présente la perte estimée due au surcoût d’ordonnancement pour chacune de ces quatre implémentations.

En ce qui concerne l’implémentation “Dédiée”, il n’y a pas de file d’attente puisque durant chaque phase d’ordonnancement chaque *thread* est lui associé une transaction dédiée. Ainsi dans le cas de deux *threads* par exemple chaque phase est

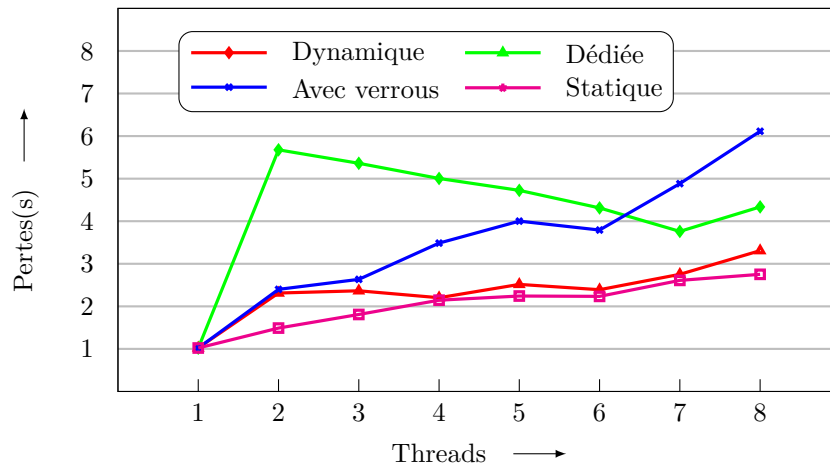


Figure 6.11 Surcoût d’ordonnancement.

composée d’au plus 2 transactions. Ceci a pour avantage d’éliminer le surcoût dû à la synchronisation de la file d’attente, mais d’un autre côté d’augmenter le nombre de phases durant chaque itération qui a pour cause d’augmenter le surcoût dû à la gestion des phases. Au fur et à mesure que le nombre de *threads* augmente chaque phase va avoir plus de transactions qui engendre une diminution de nombre de phases et ainsi une diminution du surcoût total. Lorsque le nombre de *threads* est égal à 8 le nombre de phases total se rapproche de celui de l’implémentation “Dynamique” d’où la perte perçue aussi.

Nous remarquons aussi que pour les cas de “Dynamique”, “Statique” et “Avec verrous” le surcoût est proportionnel au nombre de *threads* et augmente au fur et à mesure que le nombre de *threads* augmente. Cette augmentation est beaucoup plus prononcée dans le cas de “Avec verrous”. Ceci s’explique par le fait que le surcoût dû à la synchronisation de la file d’attente est plus important lors d’une synchronisation basée sur les verrous comparé à celui engendré lors de l’utilisation d’une synchronisation non bloquante. Ceci affirme l’avantage de l’utilisation de la synchronisation non bloquante par rapport à l’utilisation d’une synchronisation bloquante.

Maintenant il reste à déterminer l'impact de surcoût dû à la synchronisation de la file d'attente sur le surcoût total d'ordonnancement dans le cas de Dynamique ou Statique. Pour cela nous constatons que les pertes perçues en cas de "Dynamique" et "Statique" restent proches. En fait nous nous attendons à ce que "Dynamique" génère plus de perte par rapport à "Statique" parce que la file d'attente dans le cas de "Dynamique" est beaucoup plus sollicitée. Ceci montre l'efficacité de l'utilisation de la synchronisation non bloquante où le surcoût est assez faible. En effet afin d'affirmer que l'augmentation du surcoût d'ordonnancement lorsque le nombre de *threads* augmente est principalement due au surcoût de gestion des phases, nous avons comparé les résultats des tests des implémentations "Dédiée" et "Dynamique" dans le cas où le nombre de phases est le même pour les deux implémentations. Dans ce cas puisque le nombre de phases est le même alors le surcoût de gestion des phases devrait être presque égal entre les deux cas. Et puisque le cas de "Dédiée" n'engendre pas de surcoût dû à la synchronisation de la file d'attente alors il devrait générer la moindre perte due au surcoût total d'ordonnancement. Cependant les résultats des tests ont montré que les surcoûts d'ordonnancement pour les deux implémentations sont presque identiques. Ainsi nous pouvons en déduire que le surcoût dû à la synchronisation de la file d'attente lors d'une implémentation non bloquante est faible et n'a pas d'influence majeure sur la perte totale et si on voulait améliorer les performances il faudrait minimiser le traitement lors de la gestion des phases d'ordonnancement. En ce qui concerne le surcoût d'interférence les graphes de la Figure 6.12 représentent les pertes dues à ce surcoût pour les quartes implémentations décrites plus haut.

Nous remarquons que dans le cas de "Dynamique" par exemple nous avons perdu une accélération d'environ **0.5** à cause de ce surcoût d'interférence lors de l'utilisation de 8 *threads*. Afin de diagnostiquer de prêt la cause de ce surcoût nous avons utilisé l'outil d'analyse de performance, «*Concurrency Visualizer*», qui vient avec l'environnement de développement Microsoft Visual Studio 2010. Cet outil permet d'instrumenter l'exécution des applications multithreads afin de générer des informations détaillant l'interaction de ces applications avec elles-mêmes, le

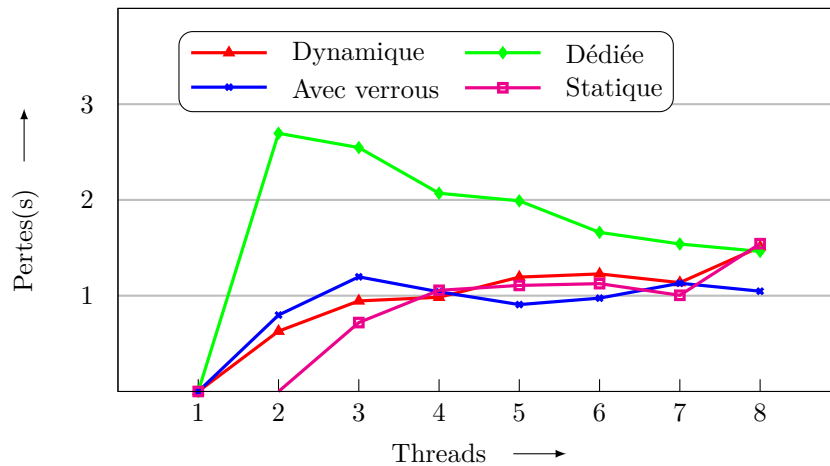


Figure 6.12 Surcoût d'interférence.

matériel, le système d'exploitation et d'autres processus sur l'ordinateur. Parmi ces informations nous retrouvons entre autres :

- Le temps écoulé pendant lequel un *thread* était en état d'exécution ou en état de blocage dû à une synchronisation, E/S, défauts de page, préemption, etc.
- Les dépendances entre les *threads*.
- Des statistiques sur la partie du code qui a engendré le plus de temps d'exécution ou qui était la cause d'un temps élevé de synchronisation

Ainsi grâce à cet outil nous avons remarqué que lors de l'exécution d'une transaction, une partie du temps écoulé est due à la synchronisation de *threads* de l'application avec les *threads* dédiés à la gestion du Garbage Collector. En effet, la Figure 6.13 illustre une situation de dépendance entre le Thread numéro 332, qui fait partie du Pool de *thread* de l'application en cours de test, et le *thread* dédié pour la gestion du Garbage Collector numéro 5072. L'état de la pile au moment du blocage (Appel de la fonction `wait_for_gc_done`) montre qu'il était en train d'attendre la fin de l'exécution du Garbage Collector. Le trait de dépendance entre les deux *threads* montre que c'est le *thread* 5072 qui a débloqué le *thread* 332

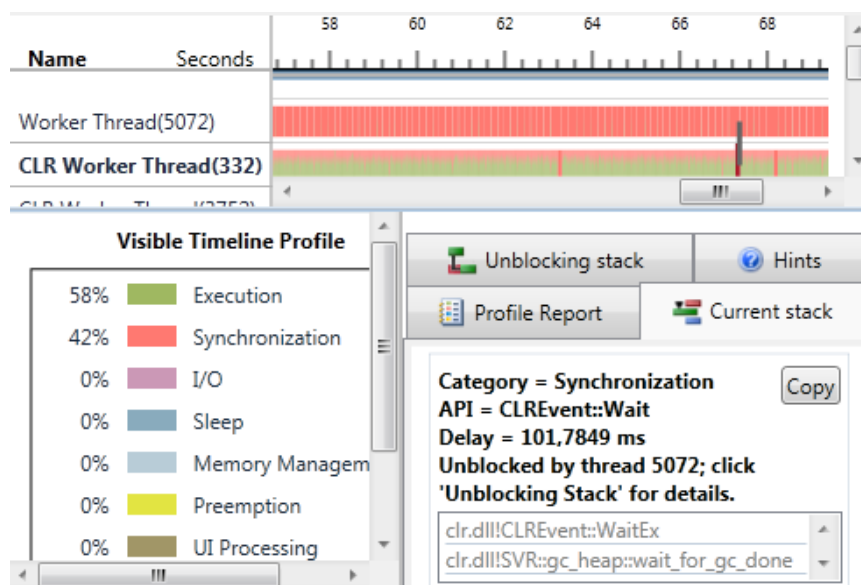


Figure 6.13 Attente du Garbage Collector pour finir le ramassage.

une fois qu'il a fini l'exécution du Garbage Collector. Pour savoir l'état de la pile du *thread* qui est à l'origine du déblocage, nous pouvons consulter l'onglet «*Unlocking stack*» (Ce n'est pas affiché dans la Figure).

Cette situation de synchronisation entre les *threads* de l'application et les *threads* du Garbage Collector est à l'origine du surcoût d'interférence dont nous avons illustré son impact dans la Figure 6.12. Les résultats de la Figure 6.12 sont obtenus en configurant un Garbage Collector de type serveur. C'est ce type de Garbage Collector qui a engendré le surcoût d'interférence le moins important. En fait, on retrouve en tout trois types de Garbage Collector que voici :

- **Station de Travail non Simultané** : Le Garbage Collector s'exécute par le *thread* qui a déclenché ce dernier. Pendant l'exécution du GC tous les autres *threads* sont suspendus en attente de la fin de la collection.
- **Station de Travail Simultané** : Ce type de Garbage Collector permet que certaines phases de la collection se déroulent dans un *thread* dédié concurrentement avec d'autres *threads* de l'application. Ceci n'est pas dans le but

d'avoir de meilleures performances, mais plutôt de minimiser la durée des pauses. Il est plutôt utile pour les applications avec interface utilisateur afin que l'utilisateur ne sente pas que l'application se bloque de temps à autre. Ainsi il favorise un meilleur temps de réponse par rapport à une meilleure gestion de la mémoire et un meilleur débit.

- **Serveur** : Un Garbage Collector de type serveur permet de paralléliser l'exécution d'une collection par plusieurs *threads*. Durant une exécution du Garbage Collector de ce type les autres *threads* de l'application sont toujours suspendus, cependant le temps d'une collection est maintenant moins importants comparé à celui des autres types puisque le traitement est partagé par plusieurs *threads* parallèles.

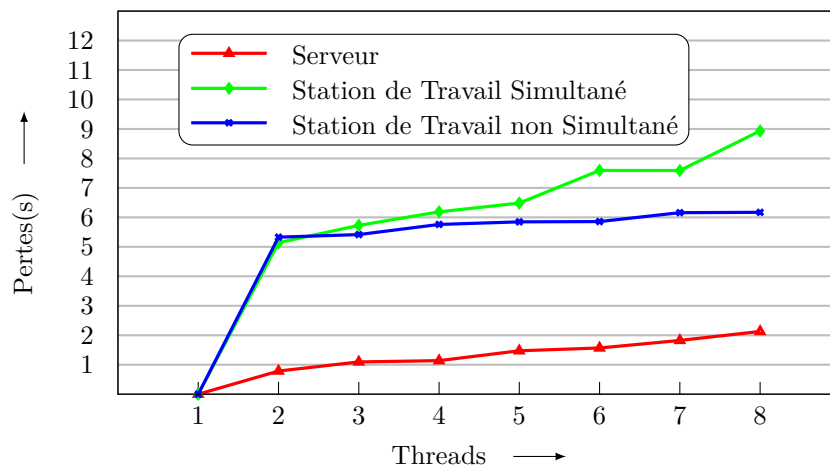


Figure 6.14 Comparaison des surcoûts d'interférence selon le type du Garbage Collector.

Le Garbage Collector de type "Station de Travail Simultané" est le type qui est pris par défaut par la plate-forme .Net. Ainsi au début des tests nous avons obtenu des accélérations qui ne dépassent pas les 2.5. Par la suite, grâce à l'outil d'analyse des performances nous avons pu détecter la principale cause du surcoût d'interférence et nous avons par conséquent passé à un Garbage Collector de type

Serveur qui nous a permis d'avoir une perte due au surcoût interférence, à peu près, 4 fois inférieure¹(voir Figure 6.14).

Afin de pouvoir minimiser encore le surcoût d'interférence il faut minimiser le traitement du Garbage Collector pour qu'il ait moins d'objets à nettoyer. Pour cela il faut diminuer la pression sur la mémoire en optimisant les allocations des objets. Ainsi en inspectant davantage le code en s'aidant de l'outil d'analyse des performances pour collecter des informations concernant les allocations et les durées de vie des objets, nous avons remarqué qu'il y a des tableaux qui peuvent être créés une seule fois et réutilisés plusieurs fois au lieu d'être instanciés à chaque fois qu'ils sont utilisés. Aussi certaines classes d'objets qui sont assez utilisées ont été remplacées par des structures. L'avantage d'utiliser des structures est que ces dernières sont allouées dans la pile d'exécution au lieu du heap. Donc elles ne sont pas sujettes au Garbage Collector. Ces changements nous ont permis de minimiser davantage le surcoût d'interférence.

6.2.4 Optimisation de l'ordonnement

D'après le théorème de sérialisabilité 4.15, pour qu'une exécution concurrente soit valide il faut que le graphe de sérialisabilité soit acyclique. Par conséquent si le multigraphe de conflit d'un modèle transactionnel est acyclique alors nous sommes sûres que n'importe quelle exécution concurrente sera valide puisque son graphe de sérialisabilité sera automatiquement acyclique. Dans ce cas notre ordonnancement va être constitué d'une seule phase composée de toutes les transactions où elles peuvent être exécutées simultanément sans risque de briser la propriété d'atomicité de chacune d'elle. Ceci a pour avantage d'augmenter le degré du parallélisme en minimisant le nombre de phases et le surcoût d'ordonnement qui s'en suit. D'où on devrait améliorer les performances.

Maintenant dans le cas où le multigraphe de conflit(voir définition 4.18), que nous notons MC , est composé de cycles, nous allons essayer d'empêcher, lors d'une

1. Ces résultats sont obtenus en utilisant Windows Vista. Lorsque nous avons mis à jour la machine à Windows 7, la différence des pertes entre les trois configurations est devenue moins importante.

exécution concurrente, de se retrouver avec un graphe de sérialisabilité cyclique. Pour cela nous allons procéder comme suit :

1. **Génération d'un graphe de verrous** : Nous commençons par calculer un arbre couvrant («*Spanning Tree*») le multigraphe du conflit que nous notons par ST . Appelons graphe de verrous le graphe résultant de la différence de MC et ST , $MC - ST$, après avoir fusionné les arêtes parallèles.
2. **Génération et Attribution Des Verrous** : Si nous ajoutons à l'arbre couvrant une arête a du graphe de verrous, alors nous allons former un cycle. Pour que tel cycle ne soit pas la cause d'un circuit dans le graphe de sérialisabilité, il faut que les transactions qui sont reliées par l'arête a soient synchronisées de telle façon qu'elles ne puissent pas s'exécuter en parallèle. Par conséquent, nous allons synchroniser l'exécution de deux transactions adjacentes par des verrous pour éviter qu'elles ne s'entrelacent. Chaque verrou est attribué à deux ou plusieurs transactions. Nous allons voir plus bas à la section 6.2.4.1 comment générer et attribuer les verrous aux transactions. Ainsi si une transaction est associée à un verrou, elle doit acquérir ce verrou avant qu'elle ne puisse entamer son exécution. Une fois l'exécution est finie, le verrou est libéré. De cette manière deux transactions qui partagent un même verrou ne peuvent pas s'entrelacer. Afin d'éviter de se retrouver dans une situation d'interblocage, les verrous doivent être obtenus toujours dans un même ordre.
3. **Exécution des transactions** : Durant chaque itération, toutes les transactions sont lancées simultanément. Le fait que certaines transactions sont synchronisées par des verrous va engendrer certaines séquentialités lors de l'exécution afin de garder la propriété de sérialisabilité comme mentionnée plus haut. La Figure 6.15 illustre l'ordonnancement des transactions appliqué en présence des verrous.

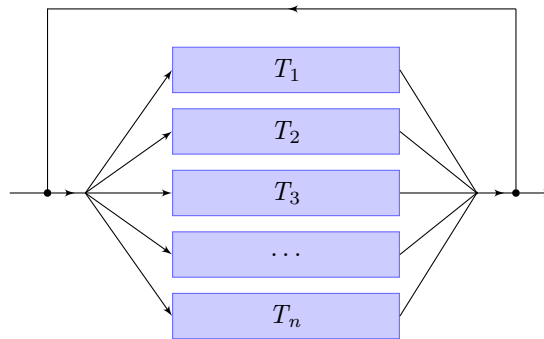


Figure 6.15 Ordonnement des transactions en présence de verrous.

6.2.4.1 Minimisation des verrous

La manière la plus directe pour générer les verrous est d'associer à chaque arête du graphe de verrous un verrou. Cependant ceci n'est pas optimal et peut générer des verrous qui ne sont pas nécessaires et qui risquent d'augmenter le surcoût de synchronisation et ainsi dégrader les performances. Prenant à titre d'exemple le graphe de la Figure 6.16.

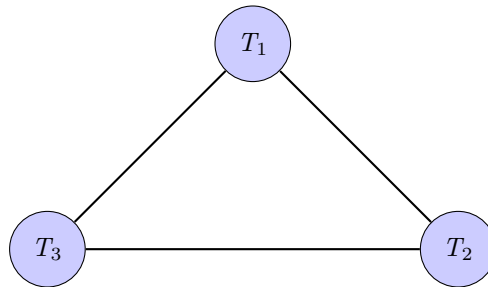


Figure 6.16 Associer un seul verrou pour les trois transactions du graphe de verrous au lieu d'un verrou pour chaque paire.

Si nous associons à chaque arête un verrou, nous allons nous retrouver avec trois verrous distincts. Cependant d'après la relation entre les trois transactions il ne peut y avoir qu'une seule transaction qui peut s'exécuter à un moment donné.

Par conséquent nous pouvons associer les trois transactions à un seul et même verrou.

Le problème de minimisation des verrous a été étudié dans [64] en proposant une heuristique étant donné que le problème est NP-complet. Pour la résolution de ce problème de minimisation dans le cas de la présente étude de cas nous avons utilisé, de la même manière que pour le calcul de l'ordonnancement qui est déjà présenté dans la section 5.3, le solveur des problèmes linéaires "IBM ILOG CPLEX Optimizer[33]". Dans ce qui suit nous présentons la formulation du problème linéaire de minimisation de verrous[64].

Étant donné un graphe de verrous $G = (V, E)$, nous introduisons des variables binaires $f_{u,i}$ pour indiquer si le verrou i est affecté au nœud u dans G , $1 \leq u \leq |V|$, $1 \leq i \leq |E|$. Nous utilisons aussi des variables binaires l_i pour indiquer si le verrou i est utilisé par au moins une transaction : $l_i = f_{1,i} \vee f_{2,i} \vee \dots \vee f_{|V|,i}$. Cette condition est représentée par les contraintes suivantes :

$$f_{1,i} + \dots + f_{|V|,i} \geq l_i \quad (6.2)$$

$$f_{1,i} + \dots + f_{|V|,i} \leq |V| \times l_i \quad (6.3)$$

Nous introduisons ensuite les variables binaires $s_{u,v,i}$ pour indiquer si les transactions u et v partagent un même verrou i . Ainsi nous avons $s_{u,v,i} = f_{u,i} \wedge f_{v,i}$. Cette condition est imposée par les contraintes suivantes :

$$f_{u,i} + f_{v,i} \geq 2 \times s_{u,v,i} \quad (6.4)$$

$$f_{u,i} + f_{v,i} \leq 2 \times s_{u,v,i} + 1 \quad (6.5)$$

Nous utilisons enfin les variables binaires $s_{u,v}$ pour indiquer si deux transactions u et v partagent au moins un verrou. Ceci est décrit par les contraintes suivantes :

$$s_{u,v,1} + \dots + s_{u,v,|E|} \geq s_{u,v} \quad (6.6)$$

$$s_{u,v,1} + \dots + s_{u,v,|E|} \leq |E| \times s_{u,v} \quad (6.7)$$

Maintenant nous voulons que deux transactions adjacentes dans le graphe de verrous partagent au moins un verrou. D'un autre côté, deux transactions non adjacentes ne doivent partager aucun verrou pour qu'elles puissent s'exécuter en parallèle. Ces deux conditions sont imposées par les contraintes suivantes :

$$s_{u,v} = \begin{cases} 1, & \text{si les transactions } u \text{ et } v \text{ sont adjacentes,} \\ 0, & \text{autrement.} \end{cases} \quad (6.8)$$

Le nombre total N de verrous utilisés est :

$$N = l_1 + \dots + l_{|E|} \quad (6.9)$$

Le problème d'optimisation linéaire est alors de minimiser N sujet aux contraintes 6.2 à 6.8.

Malheureusement la résolution du problème de minimisation des verrous en utilisant la formulation précédente prend un temps considérable dû à la présence de symétrie, comme c'était le cas pour le problème de partitionnement en utilisant le modèle des équations 5.2-5.6. Afin d'optimiser le temps de résolution, nous proposons une nouvelle formulation en nous basant sur le problème de couverture de cliques maximales. En effet, pour que nous ayons un nombre de verrous inférieur aux nombres d'arêtes il faut que nous puissions attribuer un même verrou à plus que deux transactions. Les contraintes 6.8 imposent que pour que deux transactions ou plus partagent un même verrou il faut qu'elles appartiennent à une même clique (sous graphe complet). Ainsi si nous associons à chaque clique un verrou unique, le problème d'optimisation consiste alors à minimiser le nombre de cliques nécessaires à couvrir toutes les arêtes de graphes. Et puisque pour chaque solution à ce problème nous pouvons remplacer une clique non maximale par une autre maximale englobant la première, nous pouvons nous limiter aux solutions ayant seulement des cliques maximales. Ainsi pour minimiser le nombre de verrous nous procédons comme suit.

D'abord, nous exécutons l'algorithme de Bron-Kerbosch[13] afin de trouver l'ensemble T de toutes les cliques maximales. Ensuite nous associons pour chaque

clique $t \in T$ une variable binaire x_t telle que :

$$x_t = \begin{cases} 1, & \text{si la clique } t \text{ a été attribuée un verrou.} \\ 0, & \text{si la clique } t \text{ n'a pas besoin de verrou.} \end{cases}$$

Le problème d'optimisation consiste alors à minimiser le nombre de cliques en utilisant la fonction objective suivante :

$$\min \sum_{t \in T} x_t, \quad (6.10)$$

en s'assurant que toutes les cliques qui lui ont été attribuées un verrou couvrent tout le graphe. Autrement dit, chaque arête du graphe doit appartenir à au moins une clique qui lui a été attribuée un verrou. Ceci est imposé par les contraintes suivantes :

$$\sum_{\{t \in T \mid e \in t\}} x_t \geq 1 \quad e \in E \quad (6.11)$$

Grâce à cette nouvelle formulation le temps de résolution pour l'étude de cas en cours (voir section 6.1) est passé de plusieurs minutes à quelques 1/10 de secondes.

6.2.4.2 Abstraction en variables

Avant d'introduire cette nouvelle optimisation qui consiste à abstraire certaines structures de données en variables, nous allons tout d'abord appliquer les étapes de génération de verrous sur l'étude de cas en cours (voir section 6.1) afin de mieux illustrer le besoin de cette optimisation.

La Figure 6.17 montre le multigraphe de conflit correspondant à l'étude de cas en cours en utilisant une IFFT combinatoire. Puisque chaque conflit correspond à une arête, nous pouvons nous retrouver avec une ou plusieurs arêtes reliant la même paire de transactions. Ce cas de plusieurs arêtes parallèles est représenté par le double trait. \equiv .

Puisque des arêtes parallèles forment des cycles dans le multigraphe de conflit, ces cycles vont causer des arêtes supplémentaires dans le graphe de verrous. La Fi-

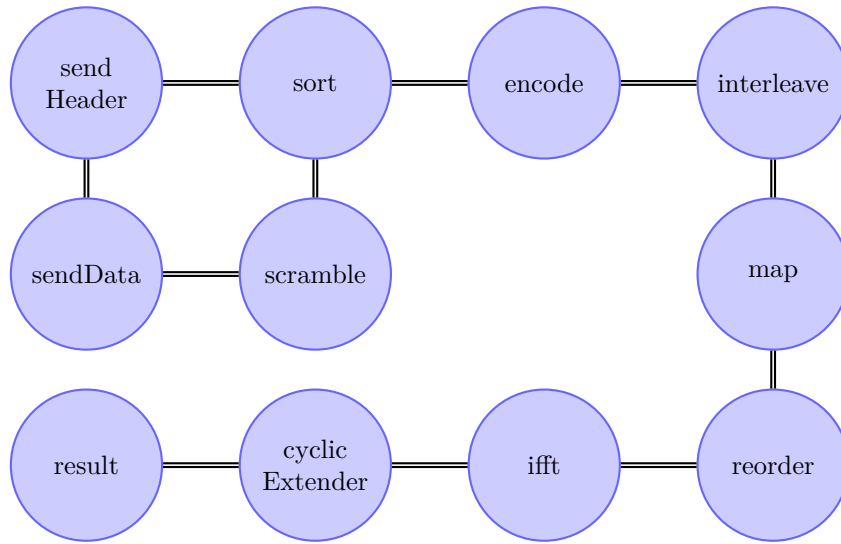


Figure 6.17 Multigraphe de Conflit

Figure 6.18 montre le graphe de verrous obtenu ainsi que les attributions de verrous à chacune des arêtes de ce graphe. Nous constatons que l'exécution des transactions synchronisées par les verrous selon la Figure 6.18 n'ajoute pas plus de parallélisme que celui obtenu lors de l'ordonnement en deux phases de la figure 6.5 où dans chaque phase les transactions peuvent s'exécuter en parallèle sans nécessité de synchronisation. En effet deux transactions sont synchronisées par un même verrou dans le graphe de verrous si et seulement si elles appartiennent à deux phases successives dans l'ordonnement de la Figure 6.5. Par conséquent si deux transactions peuvent s'exécuter en parallèle dans le premier cas alors il en est de même pour le deuxième cas et inversement. Puisque le même degré de parallélisme est obtenu dans les deux cas, les performances vont être plutôt influencées par l'importance de surcoût que chaque stratégie va engendrer.

Dans ce qui suit nous allons proposer comment minimiser le nombre de cycles du multigraphe de conflit afin d'obtenir plus de parallélisme entre les transactions. L'idée est d'abstraire certaines structures de données en de simples variables en considérant que les opérations à l'intérieur de ces structures de données correspondent à de simples opérations atomiques d'écritures et lectures. Suite à cela nous pouvons utiliser des structures de données concurrentes conçues et opti-

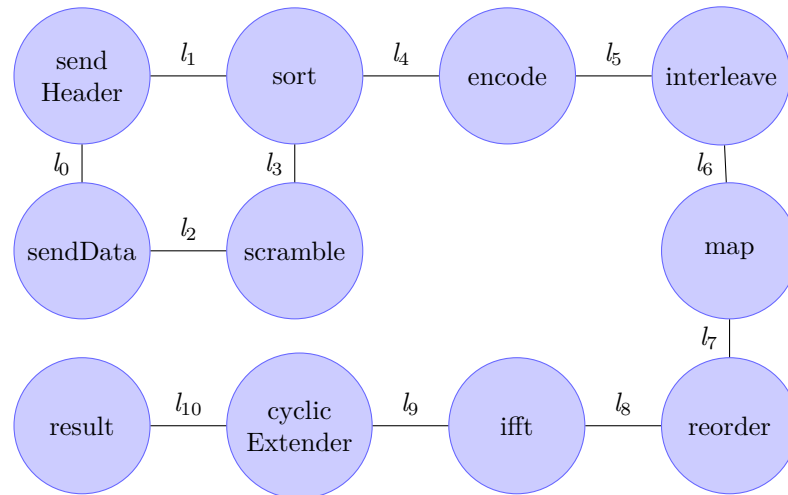


Figure 6.18 Graphe de Concurrency et Attribution des Verrous.

misées pour une exécution parallèle. Afin de mieux expliquer l'idée, nous allons illustrer celle-ci à l'aide de la FIFO. Supposons que deux transactions t_1 et t_2 communiquent à travers la FIFO, la première fait une opération `enqueue` pour enfiler une nouvelle donnée et l'autre fait une opération `dequeue` pour défiler une donnée de la FIFO. Le code suivant montre les instructions à l'intérieur de chacune de ces opérations.

```
public void enqueue(T x) {

    if (num_elements == data_size) Retry();

    data[(head + num_elements) % data_size] = x;
    num_elements++;
}
public T dequeue(){

    if (num_elements == 0) Retry();

    T val = data[head];
    head = (head + 1) % data_size;
    num_elements--;
```

```

return val;
}

```

En supposant que le seul conflit entre les deux transactions est lors de l'accès à la FIFO, le sous-multigraphe de conflit entre ces deux transactions va être composé de 4 arêtes, comme illustrer dans la Figure 6.19. Chacune de ces arêtes représente une relation de conflit de type *rw*, *wr* ou *ww* sur une variable partagée. Ce cas

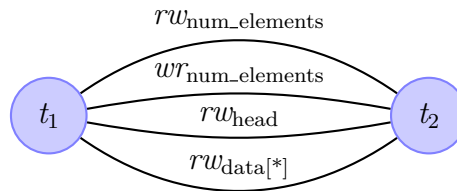


Figure 6.19 Relations de conflits lors de l'accès à la FIFO

d'accès conflictuel à la FIFO explique la présence de cycles dans le multigraphe de conflit de la Figure 6.17 entre certaines paires de transactions.

Maintenant supposons que nous avons implémenté une FIFO concurrente dont les opérations `enqueue` et `dequeue` sont atomiques et sont optimisées pour permettre un accès parallèle avec un moindre surcoût de synchronisation. Le cas de la FIFO qui a été prouvée correcte dans l'Annexe I est un exemple de FIFO concurrente que nous avons implémenté dans notre environnement .NET. L'avantage de cette implémentation est qu'elle n'a pas besoin d'utiliser des verrous ni de constructions non bloquantes pour contrôler la synchronisation en cas d'accès concurrent entre le producteur et le consommateur.. La Figure 6.20 donne un aperçu sur l'implémentation de la FIFO.

Les attributs `WriteAccess` et `ReadAccess` permettent à l'analyseur d'abstraire les opérations `enqueue`, `dequeue` et `pop` comme étant des simples opérations atomiques d'écriture et de lecture. Ceci va permettre de généraliser la notion de conflit à des structures de données en les traitant comme des simples variables où les seules opérations possibles sont l'écriture et la lecture.

```

[WriteAccess]
public T Dequeue(){
    if (!full[head]) Retry();
    T value = frame[head];
    full[head] = false;
    head = (head + 1) % _size;
    return value;
}

[ReadAccess]
public T Pop() {
    if (!full[head]) Retry();
    return data[head];
}

[WriteAccess]
public void Enqueue(T v){
    if (full[tail]) Retry();
    frame[tail] = v;
    full[tail] = true;
    tail = (tail + 1) % _size;
}

```

Figure 6.20 Exemple de FIFO Concurrente

La Figure 6.21 montre le nouveau multigraphe de conflit généré suite à cette abstraction de la FIFO en variable.

Nous remarquons que certaines arêtes parallèles ont été remplacées par une seule arête minimisant ainsi le nombre total des cycles. Ceci a eu comme conséquence de minimiser le nombre de verrous qui a eu pour avantage d'augmenter le parallélisme entre les transactions.

La Figure 6.22 compare les résultats de cette nouvelle stratégie par rapport à l'ancienne.

6.2.4.3 Évaluation des optimisations

Nous avons testé trois autres implémentations différentes afin d'évaluer les optimisations proposées précédemment dont voici les descriptions :

- **Dynamique** : C'est l'implémentation qui a donné les meilleurs résultats dans la section précédente (voir Figure 6.9). Elle utilise un pool de *threads* avec file d'attente (voir Section 5.3.1).

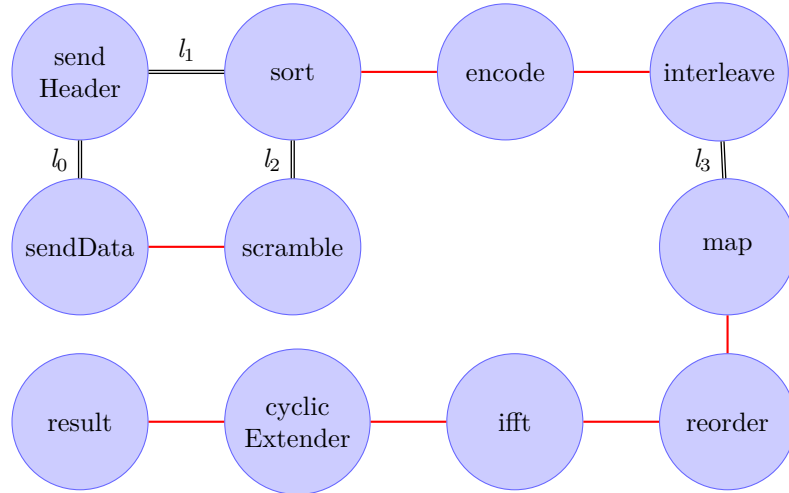


Figure 6.21 Nouveau multigraphe de conflit ayant moins de cycles et de verrous

- **Verrous** : C’est l’implémentation décrite dans la section 6.2.4. L’ordonnancement est composé d’une seule phase. Nous utilisons des verrous pour synchroniser les transactions conflictuelles et éviter de nous retrouver avec un graphe de sérialisabilité cyclique.
- **Verrous & Abstraction** : Dans cette configuration nous implémentons l’optimisation proposée dans cette section 6.2.4.2 en abstrayant la FIFO à une simple variable partagée afin de minimiser les cycles dans le graphe de conflit. Ensuite nous générons le même ordonnancement que celui de l’implémentation “Verrous” qui précède.

Dans cette expérimentation nous avons utilisé une IFFT pipelinée de trois étages. Dans chaque étage nous avons utilisé 4 blocs parallèles. La Figure 6 affiche les résultats des tests de ces trois implémentations.

Nous remarquons que les implémentations “Verrous & Abstraction” et “Verrous” sont toujours meilleures que “Dynamique”. L’explication de ce résultat est comme suit. Premièrement, même si l’implémentation “Verrous & Abstraction” nous a permis d’avoir plus de parallélisme, nous n’avons pas profité de ce parallélisme puisque le nombre de cœurs est toujours limité à 8. En plus, le parallélisme

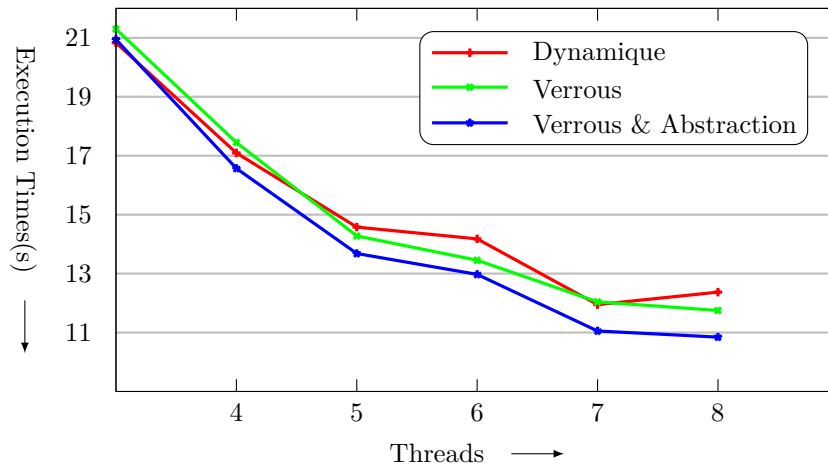


Figure 6.22 Temps d'exécution par abstraction de la FIFO

que nous avons gagné touche plutôt des transactions de faible charge de travail. Donc, pour le cas de cette expérimentation, au point de vue du parallélisme, les deux implémentations ne sont pas très différentes. Ainsi le temps que nous avons pu gagner est plutôt dû à la réduction de surcoût. En effet, l'ordonnancement dans "Verrous" et "Verrous & Abstraction" utilise une seule phase, comparée à celui de "Dynamique". Ceci a eu pour résultats de minimiser le surcoût d'ordonnancement. D'un autre côté, "Verrous" et "Verrous & Abstraction" ont un surcoût dû à la synchronisation des transactions par des verrous que "Dynamique" n'en a pas. Ce que nous pouvons conclure est que le surcoût d'ordonnancement est plus important que le surcoût de synchronisation par les verrous. Et puisque "Verrous & Abstraction" a moins de verrous que "Verrous" alors le surcoût généré est encore moindre.

Par conséquent, grâce à l'implémentation qui utilise les verrous et la FIFO optimisée nous avons amélioré l'accélération de 4.25 à **4.65**. Cependant, il nous manque toujours plus que deux points d'accélération à cause du surcoût engendré. Ainsi, il est intéressant d'appliquer de nouvelles optimisations afin de minimiser encore les surcoûts d'ordonnancement et d'interférence.

Toutefois, nous pourrions toujours améliorer l'accélération si nous avons utilisé un nombre de cœurs supérieur à huit. En effet, imaginons la situation suivante qui

n'est pas loin de la situation actuelle. Nous avons n transactions ayant à peu près le même temps d'exécution t et qui peuvent s'exécuter d'une manière totalement parallèle. Avec $n/2$ cœurs, nous pouvons avoir une accélération maximale de $n/2$. Si nous augmentons le nombre de cœurs, cette accélération ne change pas jusqu'à ce que nous arrivons à n cœurs. À ce moment l'accélération passe brusquement de $n/2$ à n et atteint sa valeur optimale. Dans notre cas, la valeur de n est égale à 14. C'est pourquoi nous remarquons (voir Figure 6.22) qu'à partir de 7 cœurs, l'accélération commence à se stabiliser. Ainsi, il faut utiliser 14 cœurs pour s'attendre à un changement. Cette situation est due au fait qu'il faut finir d'exécuter toutes les transactions avant de pouvoir passer à une autre itération. Afin de solliciter davantage les cœurs de la machine de simulation il faut penser à paralléliser les transactions sur plusieurs itérations.

Finalement, comme nouvelles voies à explorer, nous proposons un nouvel ordonnancement optimisé qui est calculé comme suit : après avoir généré le graphe de verrous, au lieu d'utiliser des verrous pour synchroniser les transactions, nous allons partitionner les transactions comme nous l'avons déjà fait pour le cas de "Dynamic" en utilisant le graphe de conflit. De cette façon, deux transactions adjacentes ne peuvent pas s'exécuter en parallèle. Ainsi nous allons obtenir un ordonnancement à plusieurs phases qui doit obligatoirement générer un meilleur temps d'exécution puisque nous nous sommes basés au départ sur un graphe de verrous qui est un sous graphe du graphe de conflit. Aussi, lorsque nous calculons l'arbre couvrant nous pouvons tenir compte du temps d'exécution des transactions pour choisir un arbre qui minimise la durée d'exécution de l'ordonnancement final à générer. Toutefois, la question qui se pose quel type de synchronisation faut-il utiliser ? Synchronisation basée sur les verrous ou utilisant les ordonnancements. Puisque notre environnement de simulation est dynamique, il est toujours possible d'estimer la meilleure configuration, puis altérer d'ordonnements au fur et à mesure que la simulation avance. Toutes ces possibilités nécessitent d'être implémenté et expérimenté. Cependant, il est plus intéressant d'utiliser une étude de cas plus importante, telle qu'un encodeur H.264 ayant un niveau d'abstraction plus haut où les charges de travail des transactions seraient plus importantes.

Conclusion

Dans un premier temps, nous présentons le bilan de cette thèse. Nous rappelons les motivations de ce travail puis nous résumons les principales réalisations et les contributions apportées. Ensuite, nous montrons comment nous pouvons étendre les travaux déjà effectués dans le but de définir une méthodologie complète basée sur les transactions permettant de réduire l'écart de productivité dans la conception des systèmes sur puce.

7.1 Bilan

Ces dernières années, il y a eu un effort de collaboration regroupant plusieurs acteurs dans l'industrie des semiconducteurs afin de soutenir et faire progresser SystemC en tant que norme pour la conception des systèmes sur puce. L'avantage de SystemC est qu'il permet de procurer une plateforme virtuelle à plusieurs niveaux d'abstraction permettant principalement l'exploration architecturale, le développement du logiciel embarqué très tôt dans le cycle de développement et la covérification logicielle/matérielle en se basant sur la simulation. Cependant, avec la complexité croissante des systèmes sur puces, d'autres besoins sont en cours d'émerger afin de réduire l'écart de productivité dans la conception de ces systèmes. L'ITRS a identifié deux défis majeurs à relever : 1) Rendre les moyens de vérification plus fiables afin de réduire l'effort de vérification et améliorer la qualité de la vérification 2) Développer des techniques de synthèse de haut niveau plus efficaces. Plusieurs efforts autour de SystemC sont en cours afin de relever ces défis tels que des travaux de recherche pour améliorer le temps de simulation, inclure des techniques de vérification basées sur les assertions, ou définir des sémantiques formelles dans le but d'appliquer la vérification formelle et la synthèse de haut

niveau. Cependant, à cause du modèle de concurrence complexe de SystemC, relever ces défis reste toujours une tâche difficile.

Par conséquent, afin de répondre aux besoins cités plus haut, nous pensons qu'il est primordial d'adopter une approche de conception qui possède une meilleure abstraction pour modéliser des composants parallèles, et ceci à plusieurs niveaux d'abstraction. Cette approche devait posséder une sémantique formelle qui constituerait la base pour développer des applications visant la vérification formelle et semi-formelle, l'optimisation de la simulation et le raffinement incrémental. Par conséquent, au cours de ce projet nous avons étudié une méthodologie de conception basée sur les transactions. Grâce à la simplicité de raisonnement que procure le concept de transaction, ce concept a eu un grand intérêt dans plusieurs domaines de l'informatique. Nous avons montré comment plusieurs de ces travaux peuvent être adaptés pour la conception des systèmes sur puce en matière de vérification formelle, raffinement incrémental et synthèse matérielle.

Dans le but d'évaluer l'efficacité de cette méthodologie, nous avons fixé l'objectif d'optimiser la vitesse de simulation d'un modèle transactionnel en profitant d'une machine multicœur. En conséquence, nous avons développé un environnement de modélisation et de simulation de systèmes sur puces basé sur le concept de transactions. Nous décrivons par la suite les principales tâches qui ont été réalisées pour arriver à bout de cet environnement et des objectifs fixés tout en précisant à chaque fois les avantages clés de cet environnement (voir Figure 7.1).

- **Modélisation** : Nous avons défini la structure à donner à un modèle transactionnel. Nous avons introduit la notion de module pour définir l'architecture d'un modèle transactionnel et la notion de transaction pour définir son comportement. Nous avons introduit aussi la construction **Retry**, une manière plus simple pour raisonner sur les gardes des transactions. Nous avons appliqué toutes ces notions pour modéliser une étude de cas industrielle spécifiant un transmetteur selon la norme Wi-Fi 802.11a.

Il est important de noter que nous avons fait en sorte que le modèle transactionnel soit complètement indépendant de la manière dont il va être simulé.

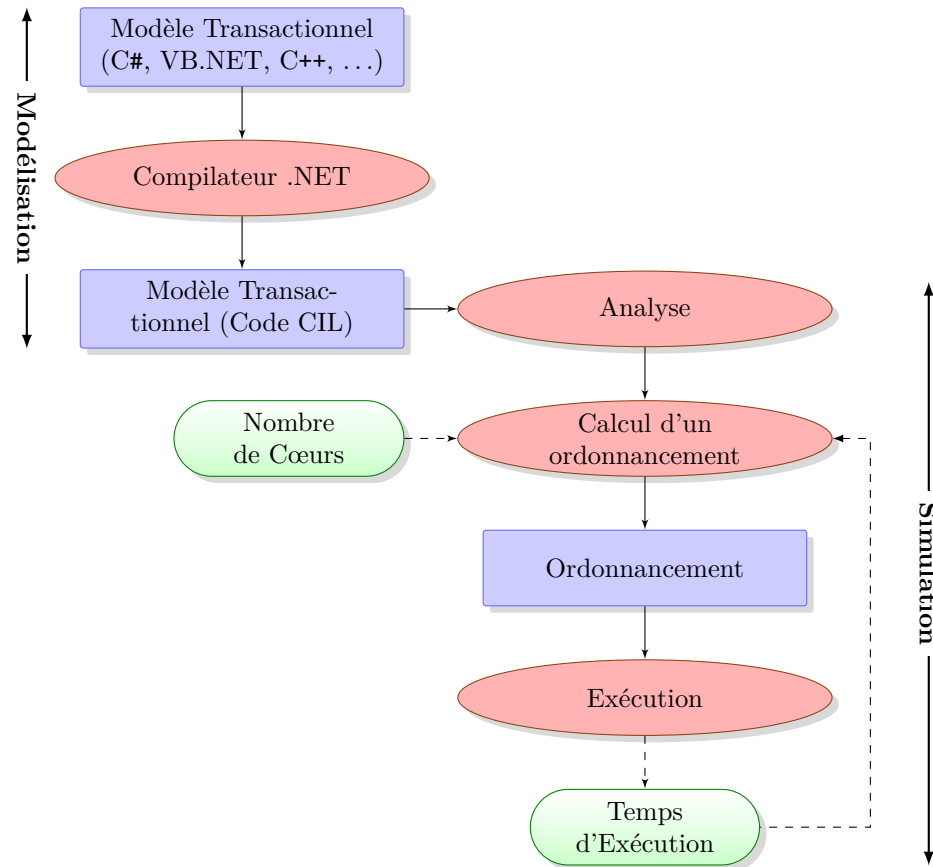


Figure 7.1 Environnement de modélisation et simulation

En effet, le modèle est représenté sous une forme de bibliothèque plutôt qu'un exécutable. Cette séparation nous a permis d'appliquer aux modèles plusieurs stratégies de simulation sans avoir à toucher ou à modifier le modèle initial.

- **Analyse** : Nous avons développé un analyseur afin d'extraire l'architecture d'un modèle transactionnel, les transactions qui le composent et l'interaction entre ces transactions. L'analyseur initie une exécution dynamique du modèle qui consiste en une phase d'élaboration afin de déterminer l'architecture du modèle à travers les modules instanciés puis d'une analyse statique pour déterminer les interactions entre les différents modules à travers les transactions déclarées. L'interopérabilité entre ces deux phases, l'une dy-

namique et l'autre statique, est facilitée grâce à l'introspection qui est une caractéristique de la plateforme .NET.

L'analyseur prend comme entrée une assemblée .NET décrivant un modèle transactionnel. Cette assemblée représente le modèle transactionnel sous un format intermédiaire en utilisant du code CIL. L'avantage de cette approche est qu'elle est complètement indépendante du langage de programmation utilisé dans le modèle initial et ainsi elle peut supporter différents langages.

Cet analyseur sera un outil essentiel pour d'autres applications en cas où nous voudrions étendre cet environnement de conception pour intégrer des solutions pour la vérification formelle ou la synthèse matérielle.

- **Ordonnement** : Après avoir donné une sémantique d'exécution formelle à un modèle transactionnel, nous avons présenté une condition nécessaire pour qu'une exécution concurrente des transactions soit considérée valide. Sur la base de cette étude théorique, nous avons implémenté des algorithmes d'ordonnement afin de pouvoir calculer des ordonnements parallèles des transactions, qui sont prouvés corrects par construction. Ces ordonnements tiennent compte du graphe de conflit du modèle transactionnel décrivant les interactions entre les différentes transactions, du nombre de cœurs de la machine de simulation et d'une estimation du temps d'exécution. Nous avons étudié différentes stratégies d'ordonnement en matière de parallélisme et de surcoût de synchronisation. Nous avons testé ces algorithmes d'ordonnement sur le modèle du transmetteur Wi-Fi 802.11a. Avec deux *threads*, nous avons obtenu une accélération d'environ 1.8 qui se rapproche d'une accélération optimum. Avec 8 *threads*, malgré que la charge de travail des différentes transactions n'était pas importante, nous avons pu atteindre une accélération d'environ 4.6, ce qui est un résultat très prometteur. En effet, si nous utilisons une étude de cas à un niveau d'abstraction plus haut où les charges de travail des transactions seraient plus importantes, nous prévoyons atteindre de meilleures accélérations.

7.2 Perspectives

La Figure 7.2 montre comment la méthodologie basée sur les transactions peut être appliquée pour la conception des systèmes sur puces. Elle représente un flot de conception complet, allant d'un modèle transactionnel de spécification fonctionnelle jusqu'à la production d'un modèle RTL prêt pour la synthèse physique.

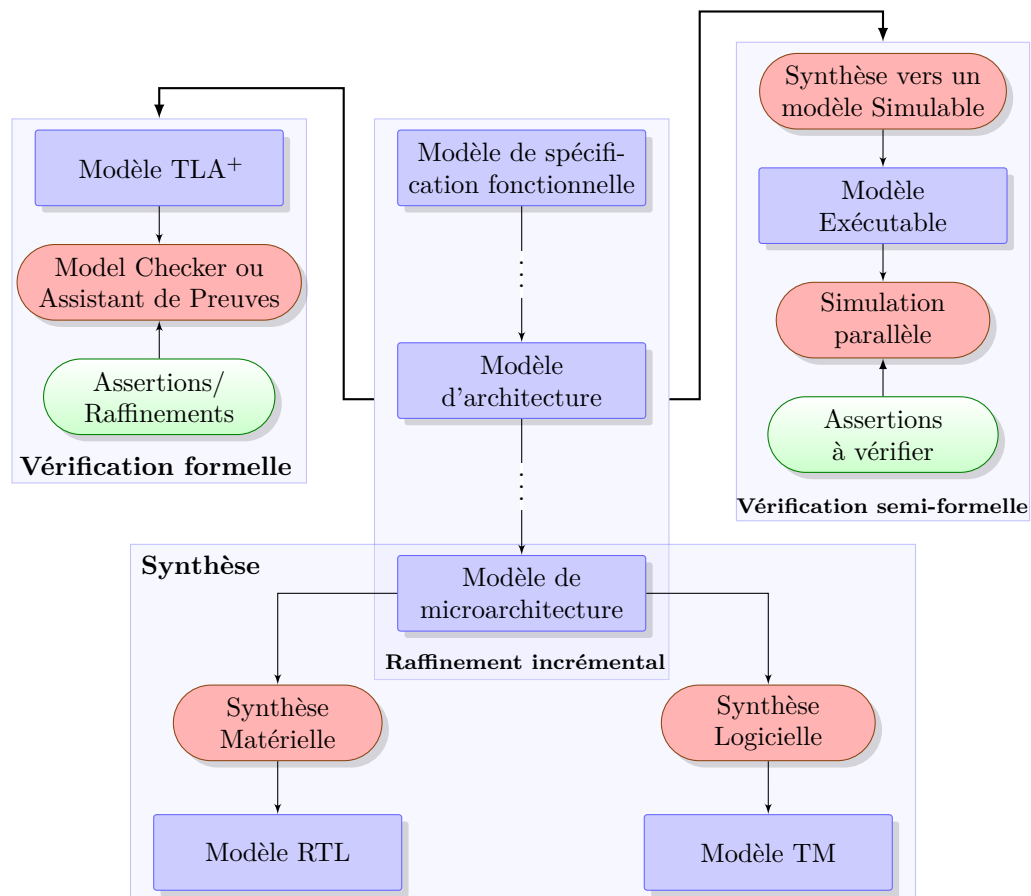


Figure 7.2 Flot de conception utilisant la méthodologie basée sur les transactions.

La clé de cette méthodologie est la conception par raffinement où le détail est ajouté et vérifié progressivement. Ceci serait possible grâce à la sémantique formelle des modèles transactionnels à partir desquels nous pouvons appliquer des techniques de vérification formelle telles que le «*Model Checking*»[39], la preuve par déduction[15] et le raffinement modulaire[1, 63] ou des techniques de vérifica-

tion semi-formelle[49] basées sur la simulation et des langages d'assertions formels tels que SVA (SystemVerilog Assertions) ou PSL (Property Specification Language).

Le flot de conception commence par spécifier un modèle de transactions décrivant les différentes fonctions constituant le système sur puce sans tenir compte de l'architecture de la plateforme d'implémentation. Chaque transaction va décrire une fonctionnalité spécifique de ce système.

Afin de vérifier la fonctionnalité du modèle de spécification, nous pouvons utiliser deux techniques sous-jacentes : une qui se base sur la vérification formelle et une qui se base sur la vérification dynamique par simulation. L'avantage de la vérification formelle est qu'elle permette une couverture complète, contrairement à la vérification par simulation qui a une couverture limitée et ne permet pas de garantir la validité du modèle. Pour les deux techniques, nous proposons d'utiliser un même langage d'assertions tels que SVA ou PSL afin de spécifier formellement les propriétés du modèle à vérifier.

En ce qui concerne la vérification formelle nous proposons deux méthodes basées sur le langage formel TLA^+ . La première est une technique de «*Model Checking*» qui se base sur le «*Model Checker*» TLC de TLA^+ . L'avantage de cette méthode est qu'elle est automatique, cependant elle ne peut généralement être appliquée qu'à des modèles de taille limitée. Toutefois, grâce à l'isolation que procure l'atomicité des transactions, certaines propriétés peuvent être vérifiées indépendamment pour chaque transaction. Par exemple, si nous vérifions que pour chaque transaction un invariant est valide, nous pouvons conclure que l'invariant reste valide pour tout le modèle. Dans le cas où la taille du modèle ne permet pas l'utilisation de la technique de «*Model Checking*» nous proposons une deuxième technique qui se base sur la preuve par déduction. L'avantage de cette technique est qu'elle n'est pas limitée par la taille du modèle, cependant, elle nécessite des connaissances approfondies des règles de déductions de la logique derrière TLA^+ . En plus, certaines preuves peuvent être longues et fastidieuses. L'application de cette technique nécessite la formulation du langage TLA^+ dans un prouveur de

théorèmes. Pour cela nous pouvons utiliser l'encodage de TLA⁺ déjà formulé à l'aide de l'assistant de preuve générique Isabelle [15].

Si la technique de vérification formelle n'arrive pas à valider certaines assertions, il est toujours possible de procéder par vérification dynamique. Pour cela les propriétés à vérifier sont transformées en des moniteurs qui s'exécutent en parallèle avec le modèle à simuler. Pour cela nous pouvons étendre les travaux faits dans [49] aux modèles transactionnels.

Au fur et à mesure que le concepteur descend dans les niveaux d'abstraction, le modèle est raffiné en ajoutant à chaque fois de plus en plus de détails décrivant l'architecture et la microarchitecture des différents composants constituant le système sur puce. Afin de montrer que le nouveau modèle raffiné est correct et vérifie les assertions déjà validées, il suffit de prouver que le raffinement fait est une implémentation du modèle de plus haut niveau. Cependant afin de pouvoir appliquer cette technique indépendamment de la complexité de design, il faut procéder d'une manière modulaire de telle manière que seulement la partie de design qui est raffinée soit vérifiée. Cette méthode est primordiale afin de contourner le problème d'explosion d'états et rendre les techniques de vérification formelle plus fiables et contrôlables, comme il a été suggéré par l'ITRS (Voir les motivations présentées plus haut 1.1). Les études déjà faites sur la composabilité et la modularité des systèmes d'actions [63] et le raffinement des spécifications TLA⁺ [1] fournissent un point d'entrée afin d'appliquer les techniques de raffinement modulaire et progressif à la conception des systèmes sur puce. Si pour une raison ou une autre le raffinement ne peut pas être validé, il est toujours possible d'appliquer les autres techniques de vérification. Ceci reste possible puisque tout le long du cycle de développement, nous utilisons le même modèle transactionnel pour décrire le système quel qu'il soit le niveau d'abstraction.

Une fois que le modèle de la microarchitecture est prêt, nous passons aux synthèses matérielles et logicielles. Pour la synthèse matérielle, il est possible de générer un modèle Bluespec ou d'étendre les travaux présentés à la section 3.2 pour générer directement un modèle RTL en VHDL ou Verilog. En ce qui concerne la synthèse logicielle, nous pensons garder le même modèle transactionnel comme im-

plémentation finale en synthétisant un gestionnaire de mémoire transactionnelle capable d'ordonnancer et synchroniser les transactions dans la plateforme cible. Cette approche qui consiste à utiliser les mémoires transactionnelles dans les programmes parallèles a été suggéré par l'ITRS, afin d'améliorer la productivité dans le développement logiciel.

Il est clair qu'établir une méthodologie de conception de systèmes sur puce basée sur le concept des transactions est un gros travail qui demande beaucoup d'efforts de recherche. Néanmoins, nous espérons que l'étude qui a été faite autour des transactions ainsi que le travail qui a été accompli durant cette thèse vont promouvoir cette méthodologie et feront continuer les recherches afin d'arriver à une meilleure méthodologie qui permettra de réduire l'écart de productivité.



Bibliographie

- [1] Martín ABADI et Leslie LAMPORT. « Conjoining specifications ». Dans : *ACM Trans. Program. Lang. Syst.* 17 (3 mai 1995), p. 507–535. ISSN : 0164-0925. DOI : <http://doi.acm.org/10.1145/203095.201069>. URL : <http://doi.acm.org/10.1145/203095.201069>.
- [2] Martín ABADI et Leslie LAMPORT. « The existence of refinement mappings ». Dans : *Theor. Comput. Sci.* 82 (2 mai 1991), p. 253–284. ISSN : 0304-3975. DOI : [http://dx.doi.org/10.1016/0304-3975\(91\)90224-P](http://dx.doi.org/10.1016/0304-3975(91)90224-P). URL : [http://dx.doi.org/10.1016/0304-3975\(91\)90224-P](http://dx.doi.org/10.1016/0304-3975(91)90224-P).
- [3] Amine ANANE, El Mostapha ABOULHAMID, Julie VACHON et Yvon SAVARIA. « Modeling and simulation of complex heterogeneous systems ». Dans : *International Symposium on Circuits and Systems*. IEEE, 2008, p. 2873–2876.
- [4] Amine ANANE, El Mostapha ABOULHAMID, Julie VACHON et Yvon SAVARIA. « Using Transaction-based Models for System Design and Simulation ». Dans : *System Level Design with .NET Technology*. CRC Press, 2009, p. 223–233.
- [5] Ralph-Johan BACK. « Refinement calculus, part II : parallel and reactive programs ». Dans : *Proceedings on Stepwise refinement of distributed systems : models, formalisms, correctness*. REX workshop. Mook, The Netherlands : Springer-Verlag New York, Inc., 1990, p. 67–93. ISBN : 0-387-52559-9. URL : <http://portal.acm.org/citation.cfm?id=91930.91938>.
- [6] Ralph-Johan BACK et Kaisa SERE. « From Action Systems to Modular Systems ». Dans : *Proceedings of the Second International Symposium of Formal Methods Europe on Industrial Benefit of Formal Methods*. FME '94.

-
- Springer-Verlag, 1994, p. 1–25. ISBN : 3-540-58555-9. URL : <http://portal.acm.org/citation.cfm?id=647536.729523>.
- [7] Ralph-Johan BACK et Kaisa SERE. « Stepwise Refinement of Action Systems ». Dans : *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*. London, UK : Springer-Verlag, 1989, p. 115–138. ISBN : 3-540-51305-1. URL : <http://portal.acm.org/citation.cfm?id=648081.746972>.
- [8] Ralph-Johan BACK et Joakim von WRIGHT. « Trace Refinement of Action Systems ». Dans : *Proceedings of the Concurrency Theory. CONCUR '94*. Springer-Verlag, 1994, p. 367–384. ISBN : 3-540-58329-7. URL : <http://portal.acm.org/citation.cfm?id=646729.703503>.
- [9] Jean BACON et Tim HARRIS. *Operating systems : concurrent and distributed software design*. UK : Addison-Wesley, 2003.
- [10] Smita BAKSHI et Daniel D. GAJSKI. « Hardware/software partitioning and pipelining ». Dans : *Proceedings of the 34th annual Design Automation Conference. DAC '97*. Anaheim, California, United States : ACM, 1997, p. 713–716. ISBN : 0-89791-920-3. DOI : <http://doi.acm.org/10.1145/266021.266349>. URL : <http://doi.acm.org/10.1145/266021.266349>.
- [11] Mike BARNETT, Manuel FÄNDRICH, Diego GARBERVETSKY et Francesco LOGOZZO. « Annotations for (more) Precise Points-to Analysis ». Dans : *IWACO 2007 : ECOOP International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*. Juil. 2007. URL : <http://publicaciones.dc.uba.ar/Publications/2007/BFGL07>.
- [12] *Bluespec*. URL : <http://www.bluespec.com/>.
- [13] Coen BRON et Joep KERBOSCH. « Algorithm 457 : finding all cliques of an undirected graph ». Dans : *Commun. ACM* 16 (9 sept. 1973), p. 575–577. ISSN : 0001-0782. DOI : <http://doi.acm.org/10.1145/362342.362367>. URL : <http://doi.acm.org/10.1145/362342.362367>.

-
- [14] Lukai CAI et Daniel GAJSKI. « Transaction level modeling : an overview ». Dans : *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. CODES+ISSS '03. Newport Beach, CA, USA : ACM, 2003, p. 19–24. ISBN : 1-58113-742-7. DOI : <http://doi.acm.org/10.1145/944645.944651>. URL : <http://doi.acm.org/10.1145/944645.944651>.
- [15] Kaustuv CHAUDHURI, Damien DOLIGEZ, Leslie LAMPORT et Stephan MERZ. « Verifying Safety Properties With the TLA+ Proof System ». Dans : *Proofs* 6173 (2010). Sous la dir. de Jürgen GIESL et ReinerEditors HÄHNLE, p. 142–148. URL : <http://arxiv.org/abs/1011.2560>.
- [16] Christophe CHEVALLAZ, Nicolas MAREAU, Frank GHENASSIA et Alain GONIER. « Advanced Methods for SoC Concurrent Engineering ». Dans : *2002 Design, Automation and Test in Europe Conference and Exposition*. IEEE Computer Society, 2002. ISBN : 0-7695-1471-5.
- [17] *Coloration de graphe*. URL : http://fr.wikipedia.org/wiki/Coloration_de_graphe.
- [18] Nirav DAVE, Michael PELLAUER et Steve GERDING. « 802.11a Transmitter : A Case Study in Microarchitectural Exploration ». Dans : *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*. ACM-IEEE, 2006.
- [19] Fredrik DEGERLUND, Marina WALDEN et Kaisa SERE. « Implementation Issues Concerning the Action Systems Formalism ». Dans : *Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies*. PDCAT '07. IEEE Computer Society, 2007, p. 471–479. ISBN : 0-7695-3049-4. DOI : <http://dx.doi.org/10.1109/PDCAT.2007.49>. URL : <http://dx.doi.org/10.1109/PDCAT.2007.49>.
- [20] Paolo DESTRO, Franco FUMMI et Graziano PRAVADELLI. « A smooth refinement flow for co-designing HW and SW threads ». Dans : *Proceedings of the conference on Design, automation and test in Europe*. DATE '07. Nice,

-
- France : EDA Consortium, 2007, p. 105–110. ISBN : 978-3-9810801-2-4. URL : <http://portal.acm.org/citation.cfm?id=1266366.1266391>.
- [21] Edsger Wybe DIJKSTRA. *A Discipline of Programming*. 1st. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1997. ISBN : 013215871X.
- [22] Piyush GARG, Sandeep K. SHUKLA et Rajesh K. GUPTA. « Efficient Usage of Concurrency Models in an Object-Oriented Co-design Framework ». Dans : *Design, Automation, and Test in Europe*. 2001.
- [23] Frank GHENASSIA. *Transaction-Level Modeling with SystemC : Tlm Concepts and Applications for Embedded Systems*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2006. ISBN : 0387262326.
- [24] Dan GROSSMAN. « The transactional memory / garbage collection analogy ». Dans : *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. OOPSLA '07. Montreal, Quebec, Canada : ACM, 2007, p. 695–706. ISBN : 978-1-59593-786-5. DOI : <http://doi.acm.org/10.1145/1297027.1297080>. URL : <http://doi.acm.org/10.1145/1297027.1297080>.
- [25] Thorsten GROTKER. *System Design with SystemC*. Norwell, MA, USA : Kluwer Academic Publishers, 2002. ISBN : 1402070721.
- [26] Ali HABIBI et Sofiene TAHAR. « Design for Verification of SystemC Transaction Level Models ». Dans : *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*. DATE '05. Washington, DC, USA : IEEE Computer Society, 2005, p. 560–565. ISBN : 0-7695-2288-2. DOI : <http://dx.doi.org/10.1109/DATE.2005.112>. URL : <http://dx.doi.org/10.1109/DATE.2005.112>.
- [27] Tim HARRIS et Keir FRASER. « Language support for lightweight transactions ». Dans : *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '03. Anaheim, California, USA : ACM, 2003, p. 388–402. ISBN :

-
- 1-58113-712-5. DOI : <http://doi.acm.org/10.1145/949305.949340>.
URL : <http://doi.acm.org/10.1145/949305.949340>.
- [28] Tim HARRIS, Simon MARLOW, Simon PEYTON-JONES et Maurice HERLIHY.
« Composable memory transactions ». Dans : *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPOPP '05. Chicago, IL, USA : ACM, 2005, p. 48–60. ISBN : 1-59593-080-9.
DOI : <http://doi.acm.org/10.1145/1065944.1065952>. URL : <http://doi.acm.org/10.1145/1065944.1065952>.
- [29] C. HELMSTETTER, F. MARANINCHI, L. MAILLET-CONTOZ et M. MOY.
« Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip ». Dans : *Proceedings of the Formal Methods in Computer Aided Design*. FMCAD '06. Washington, DC, USA : IEEE Computer Society, 2006, p. 171–178. ISBN : 0-7695-2707-8. DOI : <http://dx.doi.org/10.1109/FMCAD.2006.10>. URL : <http://dx.doi.org/10.1109/FMCAD.2006.10>.
- [30] C. A. R. HOARE. « Towards a theory of parallel programming ». Dans : New York, NY, USA : Springer-Verlag New York, Inc., 2002, p. 231–244. ISBN : 0-387-95401-5. URL : <http://portal.acm.org/citation.cfm?id=762971.762978>.
- [31] James C. HOE. « Operation-Centric Hardware Description and Synthesis ». Thèse de doct. MIT, 2000.
- [32] James C. HOE et ARVIND. « Operation-centric hardware description and synthesis ». Dans : *IEEE Trans. on CAD of Integrated Circuits and Systems* 23.9 (2004), p. 1277–1288.
- [33] *IBM ILOG CPLEX Optimizer*. URL : <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/>.
- [34] « IEEE Standard for Standard SystemC(R) Language Reference Manual ». Dans : *IEEE Std 1666-2011* (2011), p. 1–674.

-
- [35] Open SystemC INITIATIVE. *OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL*. URL : <http://www.accellera.org/downloads/standards/systemc>.
- [36] *International Technology Roadmap for Semiconductors 2007 Edition DESIGN*. URL : http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_Design.pdf.
- [37] Cedric KOCH-HOFER, Marc RENAUDIN, Yvain THONNART et Pascal VIVET. « ASC, a SystemC Extension for Modeling Asynchronous Systems, and Its Application to an Asynchronous NoC ». Dans : *Proceedings of the First International Symposium on Networks-on-Chip*. NOCS '07. Washington, DC, USA : IEEE Computer Society, 2007, p. 295–306. ISBN : 0-7695-2773-6. DOI : <http://dx.doi.org/10.1109/NOCS.2007.12>. URL : <http://dx.doi.org/10.1109/NOCS.2007.12>.
- [38] Sudipta KUNDU, Malay GANAI et Rajesh GUPTA. « Partial order reduction for scalable testing of systemC TLM designs ». Dans : *Proceedings of the 45th annual Design Automation Conference*. DAC '08. Anaheim, California : ACM, 2008, p. 936–941. ISBN : 978-1-60558-115-6. DOI : <http://doi.acm.org/10.1145/1391469.1391706>. URL : <http://doi.acm.org/10.1145/1391469.1391706>.
- [39] Leslie LAMPORT. *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN : 032114306X.
- [40] Leslie LAMPORT. « The temporal logic of actions ». Dans : *ACM Trans. Program. Lang. Syst.* 16 (3 mai 1994), p. 872–923. ISSN : 0164-0925. DOI : <http://doi.acm.org/10.1145/177492.177726>. URL : <http://doi.acm.org/10.1145/177492.177726>.
- [41] J. LAPALME, E.M. ABOULHAMID, G. NICOLESCU et F. ROUSSEAU. « Separating Modeling and Simulation Aspects in Hardware/Software System

-
- Design ». Dans : *Microelectronics, 2006. ICM '06. International Conference on*. Déc. 2006, p. 202–205. DOI : [10.1109/ICM.2006.373302](https://doi.org/10.1109/ICM.2006.373302).
- [42] James R. LARUS et Christos KOZYRAKIS. « Transactional memory ». Dans : *Communications of The ACM* 51 (7 2008), p. 80–88. DOI : [10.1145/1364782.1364800](https://doi.org/10.1145/1364782.1364800).
- [43] J.R. LARUS et R. RAJWAR. *Transactional memory*. Synthesis lectures in computer architecture. Morgan & Claypool, 2007. ISBN : 9781598291247. URL : <http://books.google.com/books?id=TuHG0EeKezoC>.
- [44] D. B. LOMET. « Process structuring, synchronization, and recovery using atomic actions ». Dans : *Proceedings of an ACM conference on Language design for reliable software*. Raleigh, North Carolina : ACM, 1977, p. 128–137. DOI : <http://doi.acm.org/10.1145/800022.808319>. URL : <http://doi.acm.org/10.1145/800022.808319>.
- [45] Enrico MALAGUTI, Michele MONACI et Paolo TOTH. « Models and heuristic algorithms for a weighted vertex coloring problem ». Dans : *Journal of Heuristics* 15 (5 oct. 2009), p. 503–526. ISSN : 1381-1231. DOI : [10.1007/s10732-008-9075-1](https://doi.org/10.1007/s10732-008-9075-1). URL : <http://portal.acm.org/citation.cfm?id=1612959.1612962>.
- [46] Kevin MARQUET, Matthieu MOY et Bageshri KARKARE. « A theoretical and experimental review of SystemC front-ends ». Dans : *Specification Design Languages (FDL 2010), 2010 Forum on*. Sept. 2010, p. 1–6. DOI : [10.1049/ic.2010.0140](https://doi.org/10.1049/ic.2010.0140).
- [47] A. MELLO, I. MAIA, A. GREINER et F. PECHEUX. « Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations ». Dans : *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*. Mar. 2010, p. 606–609.
- [48] Stephan MERZ. *An Encoding of TLA in Isabelle*.

-
- [49] M. METZGER, A. ANANE, F. ROUSSEAU, J. VACHON et E.M. ABOULHAMID. « Introspection mechanisms for runtime verification in a system-level design environment ». Dans : *Microelectronics Journal* 40.7 (2009). Mixed-Technology Testing ; Rapid System Prototyping, p. 1124–1134. ISSN : 0026-2692. DOI : [DOI:10.1016/j.mejo.2008.04.010](https://doi.org/10.1016/j.mejo.2008.04.010). URL : <http://www.sciencedirect.com/science/article/B6V44-4SRM8DP-1/2/75b91968aa1362fa17dd71edf5361e67>.
- [50] Jayadev MISRA. « A logic for concurrent programming : Safety ». Dans : *Journal of Computer and Software Engineering* 3 (1995), p. 239–72.
- [51] Wolfgang MÜLLER, Wolfgang ROSENSTIEL et Jürgen RUF, édés. *SystemC : methodologies and applications*. Norwell, MA, USA : Kluwer Academic Publishers, 2003. ISBN : 1-4020-7479-4.
- [52] Hiren D. PATEL, Sandeep K. SHUKLA, E. MEDNICK et Rishiyur S. NIKHIL. « A rule-based model of computation for SystemC : integrating SystemC and Bluespec for co-design ». Dans : *International Conference on Formal Methods and Models for Co-Design*. 2006, p. 39–48. DOI : [10.1109/MEMCOD.2006.1695899](https://doi.org/10.1109/MEMCOD.2006.1695899).
- [53] Alexandru SĂLCIANU et Martin RINARD. « Purity and Side Effect Analysis for Java Programs ». Dans : 2005, p. 199–215. URL : <http://www.springerlink.com/content/hrgarn1y96eafh1p>.
- [54] A. SALCIANU et M. RINARD. *A combined pointer and purity analysis for Java programs*. Rap. tech. MIT-CSAILTR-949. MIT, mai 2004. URL : <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.6054>.
- [55] Nick SAVOIU, Sandeep K. SHUKLA et Rajesh K. GUPTA. « Concurrency in System Level Design : Conflict Between Simulation and Synthesis Goals. » Dans : *IWLS*. 2002, p. 407–411. URL : <http://dblp.uni-trier.de/db/conf/iwls/iwls2002.html#SavoiuSG02>.
- [56] R. K. SHYAMASUNDAR, F. DOUCET, R. GUPTA et I. H. KRÜGER. « Compositional Reactive Semantics of SystemC and Verification in RuleBase ».

-
- Dans : *Proc. of the Workshop on Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. 2007.
- [57] « Supplement to IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements. Part 11 : Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications : High-Speed Physical Layer in the 5 GHz Band ». Dans : *IEEE Std 802.11a-1999* (1999), p. i. DOI : [10.1109/IEEESTD.1999.90606](https://doi.org/10.1109/IEEESTD.1999.90606).
- [58] *Thread Pool*. URL : http://en.wikipedia.org/wiki/Thread_pool_pattern.
- [59] Claus TRAULSEN, Jérôme CORNET, Matthieu MOY et Florence MARANINCHI. « A systemC/TLM semantics in PROMELA and its possible applications ». Dans : *Proceedings of the 14th international SPIN conference on Model checking software*. Berlin, Germany : Springer-Verlag, 2007, p. 204–222. ISBN : 978-3-540-73369-0. URL : <http://portal.acm.org/citation.cfm?id=1770532.1770552>.
- [60] Moshe Y. VARDI. « Formal techniques for SystemC verification ». Dans : *Proceedings of the 44th annual Design Automation Conference*. DAC '07. San Diego, California : ACM, 2007, p. 188–192. ISBN : 978-1-59593-627-1. DOI : <http://doi.acm.org/10.1145/1278480.1278527>. URL : <http://doi.acm.org/10.1145/1278480.1278527>.
- [61] Vijay V. VAZIRANI. *Approximation algorithms*. Springer, 2001, p. I–IXI, 1–378. ISBN : 978-3-540-65367-7.
- [62] Gerhard WEIKUM et Gottfried VOSSEN. *Transactional information systems : theory, algorithms, and the practice of concurrency control and recovery*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001. ISBN : 1-55860-508-8.

- [63] Xu Qiwen W. « On Compositionality in Refining Concurrent Systems ». Dans : *Proceedings of the BCS FACS 7th Refinement Workshop*. Springer Verlag, 1996.
- [64] Yuan ZHANG, Vugranam C. SREEDHAR, Weirong ZHU, Vivek SARKAR et Guang R. GAO. « Optimized lock assignment and allocation : a method for exploiting concurrency among critical sections ». Dans : *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPOPP '07. San Jose, California, USA : ACM, 2007, p. 146–147. ISBN : 978-1-59593-602-8. DOI : <http://doi.acm.org/10.1145/1229428.1229459>. URL : <http://doi.acm.org/10.1145/1229428.1229459>.

Raffinement d'une FIFO

Dans cette annexe, nous présentons comment raffiner une FIFO en utilisant le langage TLA+ et le «*Model Checker*» TLC. En commençant par une spécification abstraite, nous procédons par des étapes de raffinement progressives jusqu'à obtenir une implémentation dans un langage procédural tel que C#.

Afin de simplifier l'exemple, nous considérons le cas où la FIFO est accédée par un seul producteur et un seul consommateur. Le code suivant montre le module TLA+, *Fifo*, décrivant la spécification initiale de la FIFO.

```

MODULE Fifo
EXTENDS FifoInterface
VARIABLES seq, pc1, pc2
CONSTANTS Size

ASSUME (Size ∈ Nat) ∧ (Size > 0)

vars ≜ ⟨in, out, seq, pc1, pc2seq, pc1, pc2pc, a, b)    ≜    ∧ (pc = a)
                    ∧ (pc' = b)

Init                ≜    ∧ (seq = ⟨⟩)
                    ∧ (pc1 = 0)
                    ∧ (pc2 = 0)
                    ∧ IntInit

Enq                 ≜    ∧ ∃ v ∈ Val : IntEnq(v)
                    ∧ UNCHANGED InternalVars

ComputeEnq         ≜    ∧ Goto(pc1, 0, 1)
                    ∧ in.state = "waiting"

```

$$\begin{aligned}
& \wedge \text{Len}(\text{seq}) \leq \text{Size} \\
& \wedge \text{seq}' = \text{Append}(\text{seq}, \text{in.value}) \\
& \wedge \text{UNCHANGED} \langle \text{in}, \text{out}, \text{pc2} \rangle \\
\text{ReturnEnq} & \triangleq \wedge \text{Goto}(\text{pc1}, 1, 0) \\
& \wedge \text{IntEnqRet} \\
& \wedge \text{UNCHANGED} \langle \text{seq}, \text{pc2} \rangle \\
\text{Deq} & \triangleq \wedge \text{IntDeq} \\
& \wedge \text{UNCHANGED} \text{InternalVars} \\
\text{ComputeDeq} & \triangleq \wedge \text{out.state} = \text{"waiting"} \\
& \wedge \text{seq} \neq \langle \rangle \\
& \wedge \text{seq}' = \text{Tail}(\text{seq}) \\
& \wedge \text{out}' = [\text{out EXCEPT !.value} = \text{Head}(\text{seq})] \\
& \wedge \text{UNCHANGED} \langle \text{in}, \text{pc1} \rangle \\
\text{ReturnDeq} & \triangleq \wedge \text{Goto}(\text{pc2}, 1, 0) \\
& \wedge \text{IntDeqRet} \\
& \wedge \text{UNCHANGED} \langle \text{seq}, \text{pc1} \rangle \\
\text{Next} & \triangleq \text{Enq} \vee \text{ComputeEnq} \vee \text{ReturnEnq} \vee \text{Deq} \vee \text{ComputeDeq} \vee \text{ReturnDeq} \\
\text{FifoSpec} & \triangleq \text{Init} \wedge \square[\text{Next}]_{\text{vars}}
\end{aligned}$$

Lorsque le producteur (consommateur) est prêt pour écrire (lire) dans la FIFO il exécute l'action **Enq(Deq)**. En ce moment, l'action interne de la FIFO, **ComputeEnq(ComputeDeq)** est activée. Une fois que la FIFO a terminé son exécution, elle rend la main au producteur (consommateur) en exécutant l'action **ReturnEnq(ReturnDeq)**. La spécification complète est définie par la formule TLA **FifoSpec** qui est la disjonction des 6 actions **Enq**, **Deq**, **ComputeEnq**, **ComputeDeq**, **ReturnEnq** et **ReturnDeq**.

I.1 Interface

La spécification de la FIFO a 5 variables : **in**, **out**, **seq**, **pc1**, **pc2**. Les variables **seq**, **pc1**, **pc2** représentent l'état interne de la FIFO alors que les variables **in** et **out** représentent l'état du consommateur et du producteur durant l'appel à la

La formule **FifoSpec**, décrit les comportements (séquences de valeurs) des variables internes et externes. Cependant, un utilisateur de la FIFO interagit avec la FIFO seulement en utilisant les variables d'interface **in** et **out**. Ainsi il s'intéresse seulement aux comportements dont les variables **in** et **out** sont les seules qui sont observables. C'est-à-dire qu'une spécification de la FIFO serait de dire que les variables **in** et **out** se comportent comme s'il existe **seq**, **pc1** et **pc2** tels que le comportement des 5 variables réunies satisfait la formule **FifoSpec**. Ceci est exprimé par la formule TLA suivante :

$$\exists seq, pc1, pc2 : \text{FifoSpec}$$

I.2 Raffinement des données

Dans la spécification initiale de la FIFO nous avons utilisé la variable **seq** afin de stocker les données internes de la FIFO. Cependant si nous voulons implémenter **seq** comme un tableau, l'opération **Tail** va être couteuse en temps d'exécution puisqu'il faut décaler tous les éléments du tableau vers la gauche à chaque fois que l'opération **ComputeDeq** est exécutée. Ainsi, nous remplaçons la variable **seq** par quatre autres variables **head**, **tail**, **frame** et **full**. Les actions **ComputeEnq** et **ComputeDeq** sont alors remplacées par les définitions suivantes.

$$\begin{aligned}
 \text{ComputeEnq} &\triangleq && \wedge \text{Goto}(pc1, 0, 1) \\
 &&& \wedge in.state = \text{"waiting"} \\
 &&& \wedge full[tail] = \text{FALSE} \\
 &&& \wedge frame' = [frame \text{ EXCEPT } ![tail] = in.value] \\
 &&& \wedge full' = [full \text{ EXCEPT } ![tail] = \text{TRUE}] \\
 &&& \wedge tail' = (tail + 1) \% Size \\
 &&& \wedge \text{UNCHANGED } \langle in, out, head, pc2 \rangle \\
 \\
 \text{ComputeDeq} &\triangleq && \wedge \text{Goto}(pc2, 0, 1) \\
 &&& \wedge out.state = \text{"waiting"} \\
 &&& \wedge full[head] = \text{TRUE} \\
 &&& \wedge full' = [full \text{ EXCEPT } ![head] = \text{FALSE}]
 \end{aligned}$$

$$\begin{aligned}
&\wedge head' = (head + 1)\%Size \\
&\wedge out' = [out \text{ EXCEPT } !.value = frame[head]] \\
&\wedge \text{UNCHANGED } \langle in, frame, tail, pc1 \rangle
\end{aligned}$$

Maintenant nous voulons vérifier que la nouvelle spécification de la FIFO, `Fifo2Spec` implémente la spécification `FifoSpec`. C'est à dire vérifier que le théorème suivant est valide.

$$Fifo2Spec \Rightarrow \exists seq, pc1, pc2 : FifoSpec$$

Malheureusement, TLC n'est pas en mesure de vérifier ce genre de théorème impliquant le quantificateur existentiel. Cependant, nous pouvons définir un mapping qui lit chaque variable interne impliquant la formule `FifoSpec : seq, pc1, pc2` en fonction des variables internes de `Fifo2Spec : head, tail, frame, full, pc1, pc2`. Informellement, ceci correspond à simuler le comportement des variables `seq, pc1` et `pc2` à l'aide des variables internes de `Fifo2Spec`. Voici le module `Ref2` qui définit ce mapping :

```

┌────────────────────────────────── MODULE Ref2 ───────────────────────────────────┐
EXTENDS Fifo2

x ≜ IF head < tail THEN tail - head ELSE tail + Size - head
count ≜ IF x = Size ∧ full[head] = FALSE THEN 0 ELSE x

FifoIns ≜ INSTANCE Fifo WITH
    seq ← [n ∈ (1 .. count) ↦ frame[(head + n - 1)%Size]],
    pc1 ← pc1,
    pc2 ← pc2

Ref2 ≜ FifoIns!FifoSpec
└────────────────────────────────── THEOREM Fifo2Spec ⇒ Ref2 ───────────────────────────────────┘

```

I.3 Raffinement de l'atomicité

L'implémentation `Fifo2Spec` comprend les actions `ComputeEnq` et `ComputeDeq` qui ne peuvent être directement implémentées dans un langage procédural (sauf

en utilisant des mécanismes de synchronisation tels que l'exclusion mutuelle) puisqu'elles doivent être exécutées d'une manière atomique. Par conséquent, il faut décomposer ces actions en plusieurs actions atomiques qui correspondront à des instructions atomiques du langage cible. Voici les définitions des actions **ComputeEq** et **ComputeDeq** après décomposition.

$$\begin{aligned}
 \text{ComputeEq} &\triangleq \vee \quad \wedge \text{Goto}(pc1, 0, 1) \\
 &\quad \wedge in.state = \text{"waiting"} \\
 &\quad \wedge full[tail] = \text{FALSE} \\
 &\quad \wedge frame' = [frame \text{ EXCEPT } ![tail] = in.value] \\
 &\quad \wedge \text{UNCHANGED } \langle in, out, head, pc2, full, tail \rangle \\
 &\vee \quad \wedge \text{Goto}(pc1, 1, 2) \\
 &\quad \wedge full' = [full \text{ EXCEPT } ![tail] = \text{TRUE}] \\
 &\quad \wedge \text{UNCHANGED } \langle in, out, head, pc2, frame, tail \rangle \\
 &\vee \quad \wedge \text{Goto}(pc1, 2, 3) \\
 &\quad \wedge tail' = (tail + 1)\%Size \\
 &\quad \wedge \text{UNCHANGED } \langle in, out, head, pc2, frame, full \rangle \\
 \\
 \text{ComputeDeq} &\triangleq \vee \quad \wedge \text{Goto}(pc2, 0, 1) \\
 &\quad \wedge out.state = \text{"waiting"} \\
 &\quad \wedge full[head] = \text{TRUE} \\
 &\quad \wedge out' = [out \text{ EXCEPT } !.value = frame[head]] \\
 &\quad \wedge \text{UNCHANGED } \langle in, frame, tail, pc1, full, head \rangle \\
 &\vee \quad \wedge \text{Goto}(pc2, 1, 2) \\
 &\quad \wedge full' = [full \text{ EXCEPT } ![head] = \text{FALSE}] \\
 &\quad \wedge \text{UNCHANGED } \langle in, frame, tail, pc1, head, out \rangle \\
 &\vee \quad \wedge \text{Goto}(pc2, 2, 3) \\
 &\quad \wedge head' = (head + 1)\%Size \\
 &\quad \wedge \text{UNCHANGED } \langle in, frame, tail, pc1, full, out \rangle
 \end{aligned}$$

Afin de vérifier que la nouvelle spécification **Fifo3Spec** implémente **Fifo2Spec**, nous procédons de la même manière que pour le raffinement, **Ref2**. Voici le module **Ref3** qui permet à TLC de vérifier que la décomposition de l'atomicité est valide.

```

MODULE Ref3
EXTENDS Fifo3

Fifo2Ins ≜ INSTANCE Fifo2 WITH
  frame ← frame,

```

```

head ← IF pc2 ∈ {1, 2} THEN (head + 1)%Size ELSE head,
tail ← IF pc1 ∈ {1, 2} THEN (tail + 1)%Size ELSE tail,
full ← IF pc1 = 1 ∧ pc2 = 1 THEN
    [full EXCEPT ![tail] = TRUE, ![head] = FALSE]
ELSE IF pc1 = 1 THEN
    [full EXCEPT ![tail] = TRUE]
    ELSE
    IF pc2 = 1 THEN
        [full EXCEPT ![head] = FALSE]
    ELSE
        full,
pc1 ← IF pc1 ∈ {2, 3} THEN 1 ELSE pc1,
pc2 ← IF pc2 ∈ {2, 3} THEN 1 ELSE pc2

```

Ref3 \triangleq *Fifo2Ins!Fifo2Spec*

THEOREM *Fifo3Spec* \Rightarrow *Ref3*

I.4 Implémentation finale

La spécification de la FIFO *Fifo3Spec* contient encore certains inconvénients. En effet, la condition `full[tail] = false` est traduite par la boucle `while (full[tail] != false)`. C'est une attente active que nous voulons éviter. En prenant comme cible le langage **C#**, nous allons utiliser la classe **AutoResetEvent** afin de synchroniser les opérations de lecture et écriture. Cette classe contient deux méthodes **Set** et **WaitOn**. Si un processus veut se suspendre en attente d'un évènement, il appelle la méthode **waitOn**. Lorsqu'un autre processus appelle la méthode **Set**, le processus en suspension est notifié pour continuer son exécution. Remarquer que l'appel à la fonction **Set** est persistant. C'est-à-dire que si un processus appelle **Set** avant qu'un autre processus appelle la fonction **waitOn**, l'effet de **Set** reste visible jusqu'à qu'un processus exécute la méthode **waitOn**. À ce moment, ce dernier processus continue son exécution sans avoir à attendre. Aussi, le fait d'appeler la méthode **Set** plusieurs fois avant un appel à **waitOn** est équivalent à un seul appel à **Set**. Le module TLA+ suivant spécifie ces deux méthodes.

MODULE *AutoResetEvent*

Set(*x*) \triangleq $\wedge x' = 1$

WaitOn(*x*) \triangleq $\wedge x = 1$

$$\wedge x' = 0$$

La nouvelle spécification de la FIFO introduit les nouvelles variables `waitRead` et `waitWrite` qui vont correspondre à deux objets de classe `AutoResetEvent`. La variable `waitRead` est utilisée pour que le consommateur notifie le producteur en cas de lecture. De même, `waitWrite` est utilisée pour que le producteur notifie le consommateur en cas d'écriture. Voici la nouvelle spécification, `Fifo4Spec`, des actions `ComputeEnq` et `ComputeDeq`.

```

ComputeEnq      ≜
∨    ∧ in.state = "waiting"
      ∧ IF full[tail] = FALSE THEN
          ∧ Goto(pc1, 0, 6)
          ∧ UNCHANGED exceptpc1
      ELSE
          ∧ Goto(pc1, 0, 5)
          ∧ UNCHANGED exceptpc1
∨    ∧ Goto(pc1, 5, 0)
      ∧ WaitOn(WaitRead)
      ∧ UNCHANGED ⟨in, out, frame, full, tail, head, pc2, WaitWrite⟩
∨    ∧ Goto(pc1, 6, 1)
      ∧ frame' = [frame EXCEPT ![tail] = in.value]
      ∧ UNCHANGED ⟨in, out, head, pc2, full, tail, WaitRead, WaitWrite⟩
∨    ∧ Goto(pc1, 1, 2)
      ∧ full' = [full EXCEPT ![tail] = TRUE]
      ∧ UNCHANGED ⟨in, out, head, pc2, frame, tail, WaitRead, WaitWrite⟩
∨    ∧ Goto(pc1, 2, 3)
      ∧ Set(WaitWrite)
      ∧ UNCHANGED ⟨in, out, frame, full, tail, head, pc2, WaitRead⟩
∨    ∧ Goto(pc1, 3, 4)
      ∧ tail' = (tail + 1)%Size
      ∧ UNCHANGED ⟨in, out, head, pc2, frame, full, WaitRead, WaitWrite⟩

ComputeDeq     ≜
∨    ∧ out.state = "waiting"
      ∧ IF full[head] = TRUE THEN
          ∧ Goto(pc2, 0, 6)

```

```

      ∧ UNCHANGED exceptpc2
    ELSE
      ∧ Goto(pc2, 0, 5)
      ∧ UNCHANGED exceptpc2
  ∨   ∧ Goto(pc2, 5, 0)
      ∧ WaitOn(WaitWrite)
      ∧ UNCHANGED  $\langle in, out, frame, full, tail, head, pc1, WaitRead \rangle$ 
  ∨   ∧ Goto(pc2, 6, 1)
      ∧  $out' = [out \text{ EXCEPT } !.value = frame[head]]$ 
      ∧ UNCHANGED  $\langle in, frame, tail, pc1, full, head, WaitRead, WaitWrite \rangle$ 
  ∨   ∧ Goto(pc2, 1, 2)
      ∧  $full' = [full \text{ EXCEPT } ![head] = FALSE]$ 
      ∧ UNCHANGED  $\langle in, frame, tail, pc1, head, out, WaitRead, WaitWrite \rangle$ 
  ∨   ∧ Goto(pc2, 2, 3)
      ∧ Set(WaitRead)
      ∧ UNCHANGED  $\langle in, out, frame, full, tail, head, pc1, WaitWrite \rangle$ 
  ∨   ∧ Goto(pc2, 3, 4)
      ∧  $head' = (head + 1)\%Size$ 
      ∧ UNCHANGED  $\langle in, frame, tail, pc1, full, out, WaitRead, WaitWrite \rangle$ 

```

Voici le «*refinement mapping*» afin de valider que la nouvelle spécification **Fifo4Spec** implémente la spécification **Fifo3Spec**

```

MODULE Ref4
EXTENDS Fifo4

Fifo3Ins  $\hat{=}$  INSTANCE Fifo3 WITH
  head  $\leftarrow$  IF pc2 = 3 THEN (head + 1)%Size ELSE head,
  tail  $\leftarrow$  IF pc1 = 3 THEN (tail + 1)%Size ELSE tail,
  pc1  $\leftarrow$  IF pc1  $\in$  {5, 6} THEN 0 ELSE IF pc1 = 4 THEN 3 ELSE pc1,
  pc2  $\leftarrow$  IF pc2  $\in$  {5, 6} THEN 0 ELSE IF pc2 = 4 THEN 3 ELSE pc2

Ref4  $\hat{=}$  Fifo3Ins!Fifo3Spec

```

```

THEOREM Fifo4Spec  $\Rightarrow$  Ref4

```

Voici l'implémentation C# de la spécification **Fifo4Spec**. L'avantage de cette implémentation est qu'elle n'utilise aucun mécanisme de blocage entre les pro-

cessus producteur et consommateur. Les deux processus peuvent s'exécuter en parallèle et l'effet de leur exécution correspond à une exécution de la spécification initiale `FifoSpec`.

```
public class Fifo<T>{
    . . .
    public T Dequeue(){
        value;
        /*0*/ while (!full[head]) /*5*/ WaitWrite.WaitOne();
        /*6*/ value = frame[head];
        /*1*/ full[head] = false;
        /*2*/ WaitRead.Set();
        /*3*/ head = (head + 1) % _size;
        /*4*/ return value;
    }
    public void Enqueue(T v){
        /*0*/ while (full[tail]) /*5*/ WaitRead.WaitOne();
        /*6*/ frame[tail] = v;
        /*1*/ full[tail] = true;
        /*2*/ WaitWrite.Set();
        /*3*/ tail = (tail + 1) % _size;
        /*4*/ return;
    }
    . . .
}
```

Les numéros correspondent aux valeurs de `pc1` et `pc2` dans la spécification TLA+.