Université de Montréal

Méthodologie de conception d'un modèle comportemental pour la vérification formelle

par
Frédéric Bastien

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

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M. Sc.) en Informatique

Décembre, 2006

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Méthodologie de conception d'un modèle comportemental pour la vérification formelle

présenté par :

Frédéric Bastien

a été évalué par un jury composé des personnes suivantes :

Stefan Monnier
président-rapporteur

El Mostapha Aboulhamid
directeur de recherche

Julie Vachon
codirectrice

François Raymond-Boyer
codirecteur

Yann-Gaël Guéhéneuc
membre du jury

Mémoire accepté le 27 février 2007

Résumé

L'objectif du travail présenté dans ce document est la vérification formelle de la spécification textuelle d'un système électronique. Pour ce faire, on construit un modèle comportemental à partir de la spécification textuelle, dans une représentation (réseaux de Petri) qui permet de vérifier de façon formelle la complétude interne et la cohérence du modèle et donc de la spécification. De plus, la méthode et les outils associés permettent au concepteur de définir de nouvelles propriétés à vérifier. La méthodologie est validée sur une étude de cas portant sur la spécification du bus AMBA AHB-Lite. Grâce à la méthodologie, on a trouvé un signal avec deux définitions dont le choix de la définition influence le comportement du bus.

Mots-clefs : Systèmes électroniques, model checking, vérification de modèle, vérification formelle, réseau de Petri, réseau de Petri coloré, CPN Tools, graphe d'états, graphe de composantes fortement connexes

Abstract

The objective of the work presented in this document is the formal verification of textual specification of electronic system. To do this, we built a behavioural model from the textual specification in a representation (Petri net) that allows the verification of the completeness and coherence of the model, thus of the specification. Also, the method and the tools allow the modeller to add new properties to verify. The methodology is validated on a case study of the AHB-Lite bus of the AMBA specification. With the methodology, we found a signal with two definitions.The choice of the definition has an impact on the behaviour of the bus.

Keywords: Electronic system, model checking, model verification, formal verification, colored Petri net, CPN Tools, state space graph, strongly connected component graph

Table des matières

Liste des figures

Liste des tableaux

Remerciements

Je tiens à remercier mon directeur El Mostapha Aboulhamid et mes deux codirecteurs Julie Vachon et François Raymond-Boyer pour leur direction et leur support tout au long de ma maîtrise.

Je tiens aussi à remercier Frédéric Rousseau qui a passé beaucoup de temps à me montrer comment rédiger un mémoire, j'en avais bien besoin.

Je dois aussi remercier le ReSMiQ et le DIRO pour leur soutien financier pendant ma maîtrise.

Finalement, je tiens à remercier ma femme Isabelle pour ses encouragements et son appui dans les moments difficiles.

Introduction

Les systèmes électroniques sont de plus en plus présents dans notre vie quotidienne, notamment à travers les systèmes multimédias (les enregistreurs numériques pour la télévision, les consoles de jeux vidéo, les cellulaires couleurs). Ces différents systèmes offrent de nombreuses fonctionnalités et leur conception nécessite plusieurs composants, dont plusieurs microprocesseurs. Ces systèmes manipulent un flot de données important entre autres pour les informations audio et vidéo.

Ces besoins en traitement d'informations nécessitent à la fois plus de composants et une plus grande complexité de ceux-ci. Ce qui explique que plusieurs millions de transistors composent les systèmes intégrés. Bien entendu, cette complexité augmente le temps de conception, ce qui ralentit la mise en marché d'un nouveau système.

De plus, découvrir un bogue dans le système intégré après la fabrication coûte extrêmement cher. En effet, on évalue en ce moment qu'un jeu de masques coûte environ 0,5 million $US à produire. Et on se rend ainsi compte du coût de fabrication d'une nouvelle version d'un système.

On comprend ainsi le besoin de vérification à toutes les étapes de conception pour s'assurer que le système obtenu est bien conforme aux spécifications. Actuellement, la vérification s'effectue principalement pendant la conception du système et lorsque le système est construit pour détecter les erreurs de fabrication. Par contre, la vérification au niveau de la spécification est souvent négligée. À cette étape, il importe généralement de vérifier la complétude interne, la cohérence de la spécification et les propriétés souhaitées du système. La complétude est la propriété d'une spécification qui définit tous les éléments et sous-cas qu'elle utilise voire même indirectement (complétude interne). La cohérence est la propriété d'une spécification qui ne comporte pas d'erreurs de logique, ni de contradictions. Le travail présenté dans ce mémoire vise la mise en œuvre d'une méthode de vérification formelle pour ces deux types de propriétés.

1.1. Conséquences d'erreurs dans les systèmes

Les coûts et les conséquences qu'entraîne la découverte d'une erreur pour la compagnie qui fabrique le système comptent parmi les raisons justifiant le développement de méthodes de

vérification plus systématiques et rigoureuses. Les conséquences peuvent être de natures variables : perte de vies humaines, pertes financières, etc. On peut citer quelques exemples de systèmes dont les conséquences ont été catastrophiques : une erreur dans la division en virgule flottante du processeur Pentium a coûté 475 millions $US à la compagnie Intel [2]. La fusée Ariane 5 a explosé en vol en 1996 à cause d'une erreur logicielle [3]. Le coût estimé de la cargaison était de 400 millions d'euros et le retard de la version finale de la fusée a engendré des frais importants. Dans le cas du système Therac-25 [4] , un mauvais calcul du dosage de rayon X a engendré la perte de vies humaines.

Ceci démontre l'importance de vérifier le plus complètement possible un système avant sa mise en marché ou son utilisation.

Certes, la vérification est déjà faite dans les phases de conception des composants matériels et logiciels. On vérifie qu'ils sont conformes aux spécifications tout au long de la conception. Pour cela on utilise principalement les techniques de simulation. Mais, ces techniques ne permettent pas une vérification exhaustive de tous les états du système.

1.2. Flot de conception et de vérification des systèmes

Un système intégré est composé de deux parties : une partie matérielle et une partie logicielle. Cette dernière est un programme écrit spécifiquement pour fonctionner avec la partie matérielle. Celui-ci utilise un ou plusieurs processeurs et possiblement des accélérateurs matériels. Il est souvent plus avantageux d'utiliser un mélange d'implémentation entre le logiciel et le matériel. Ce découpage entre logiciel et matériel est déterminant en termes de performance, mais aussi d'un point de vue coût et évolution. C'est ce partitionnement qui va donner un meilleur résultat s'il est bien choisi.

Le flot standard de conception (Figure 1) d'un système part généralement d'une spécification textuelle. La première étape est de transformer cette spécification textuelle en un modèle comportemental en vue de son éventuelle simulation pour détecter et corriger des erreurs. Avec cette première version, on raffine ce modèle, c'est-à-dire on ajoute des informations et des détails pour tendre vers une réalisation physique. Le partitionnement est une des étapes de raffinement qui consiste à décider quelles parties du système seront réalisées en matériel et en logiciel.

Après le partitionnement, il y a deux flots de développement en parallèle : celui du matériel et celui du logiciel. Dans le flot du matériel, la première étape consiste à concevoir le matériel dans un

langage de description de matériel (*HDL, Hardware Description Language*). Ensuite, on synthétise le circuit pour obtenir une description du comportement au niveau matériel. Puis, on exécute le placement et le routage. Le placement et le routage positionnent les cellules de base qui contiennent les composantes matérielles et réalisent les connexions entre ces cellules. On peut alors produire les masques et lancer le circuit en fabrication.

Dans le flot de développement logiciel, on crée le logiciel. Ce flot est composé des étapes d'implémentation du logiciel, de test et de vérification de l'implémentation. Ensuite, on intègre les deux parties dans le système final.

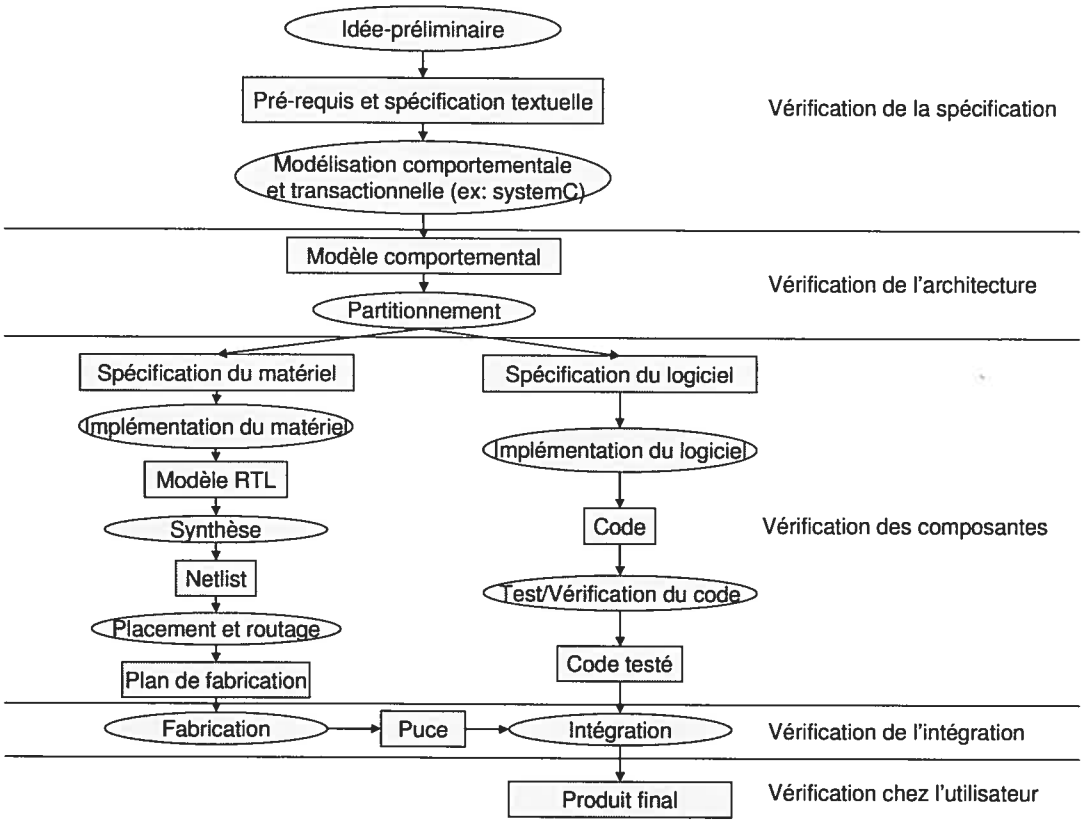

Figure 1 : Flot de conception standard

Il est couramment reconnu que la vérification prend 70 % du temps total de conception. À chaque étape, on peut trouver et corriger des erreurs introduites au cours des étapes précédentes comme on peut en ajouter de nouvelles. Plus le temps écoulé est long entre l'erreur et sa découverte, plus l'erreur est difficile à corriger et donc coûteuse.

En ce moment, il existe plusieurs techniques pour vérifier les circuits à chaque étape du cycle de développement. Ces techniques sont formelles ou non formelles. Entre autres, la simulation est une technique non formelle. Les techniques formelles donnent des preuves de certaines propriétés. Les techniques non formelles n'en donnent pas.

Un système est dit valide si les résultats produits en fonction des valeurs d'entrée sont corrects par rapport à une spécification.

La vérification formelle est une catégorie de techniques qui semble plus efficace pour la vérification des systèmes, car elle garantit qu'un modèle de circuit est valide par des preuves mathématiques. Ces techniques peuvent être utilisées à toutes les étapes de la conception du système. Les premières approches formelles utilisées nécessitaient les services de spécialistes pour élaborer des preuves et prenaient beaucoup de temps. Cela rendait ces techniques coûteuses, donc difficilement utilisables. Aujourd'hui, des logiciels d'aide à la preuve, dont certains sont d'ailleurs commerciaux, permettent d'abstraire certains détails et de faciliter le travail de preuve. Du coup, certaines catégories de problèmes peuvent être résolues par des non-spécialistes de la preuve formelle.

1.3. Étapes de vérification

Après la découverte d'une erreur, suit une séquence d'étapes pour parvenir à sa correction. Le tout est d'abord déclenché par la découverte d'un comportement non valide. Il faut alors analyser l'erreur et tenter d'en trouver la cause. Il ne reste plus qu'à appliquer le correctif. Il faut garder ces étapes en tête quand on conçoit une méthodologie de vérification, car les techniques utilisées pourront avoir un impact sur leur durée et leur difficulté de réalisation.

Certaines étapes peuvent être plus ou moins faciles selon le choix de la technique de vérification utilisée. Par exemple, les techniques par simulation semi-formelle et par prototypage permettent d'identifier certaines séquences d'entrées de longueur quelconque qui déclenchent une erreur. En revanche, les techniques formelles permettent d'identifier une séquence de taille minimale qui déclenche une erreur. Or, une séquence minimale facilite la découverte de l'endroit et de la cause de l'erreur.

Il y a plusieurs aspects à vérifier pendant la conception d'un système. Voici les principales :

- vérification de la spécification fonctionnelle;
- vérification de l'architecture du système;
- vérification de l'implémentation des composants;
- vérification de l'intégration des composants;
- vérification finale du système dans l'environnement de l'utilisateur.

La méthodologie proposée dans ce mémoire englobe la vérification de la spécification fonctionnelle et peut être étendue à la vérification de l'architecture du système. Selon le niveau d'abstraction du modèle, elle peut inclure une portion plus ou moins grande de la vérification architecturale du système.

1.4. Technique de vérification des systèmes

Au cours des ans, plusieurs techniques de vérification ont été développées. Elles sont séparées en trois catégories :

1) Les approches empiriques
   a. Le test (référent = implémentation) : exécution et observation de l'implémentation du système
   b. La simulation (référent = modèle) : exécution et observation d'un modèle logiciel du système
   c. Prototypage [5] (référent = prototype matériel) : exécution et observation d'un prototype de l'implémentation du système sur une plateforme matérielle. Par exemple sur un FPGA (émulation).
2) Les approches purement analytiques (*Theorem proving*) : font appel à un vérificateur de théorèmes et incluent les sous-catégories suivantes :
   a. Comparaison avec une version de référence;
   b. Déduction logique : on modélise le système dans un langage qui est basé sur les mathématiques. Ensuite, il s'agit de trouver toutes les étapes de la preuve en partant des axiomes et cela, dans le but de vérifier une propriété. Il y a deux méthodes de preuve

utilisées :

i. Automatique [6] : l'ordinateur fait la preuve de façon automatique. Ceci est envisageable seulement dans le cas des systèmes de taille réduite et faisant appel à des langages de modélisation de bas niveau.

ii. Assistant de preuve [7, 8] : le concepteur a la responsabilité de créer la preuve. Un logiciel vérifie cette preuve et parfois complète certaines parties de la preuve.

3) Les approches par abstraction : vérification d'un modèle formel de plus haut niveau que le système

a. Model checking [9, 10] : consiste à exprimer le système, matériel ou logiciel, par une machine à états finis et la propriété qu'on souhaite vérifier par une formule approprié. La vérification de la propriété est faite automatiquement et soit donne une preuve ou un contre-exemple de la propriété.

b. Interprétation abstraite : théorie d'approximation de la sémantique de programmes informatiques basée sur les fonctions monotones pour ensembles ordonnés, en particulier les treillis (en anglais : lattice). Elle peut être définie comme une exécution partielle d'un programme pour obtenir des informations sur sa sémantique (par exemple, sa structure de contrôle, son flot de données) sans avoir à en faire le traitement complet.[1]

c. Exécution symbolique : une simulation ou on considère plusieurs exécution en même temps. On peut utiliser une variable symbolique dans l'état du système pour représenter plusieurs états en même temps. Pour chaque valeur possible de la variable, il y a un état qui est simulé indirectement.

Les techniques de vérification ont besoin de plusieurs informations pour fonctionner. À défaut de se baser sur une implémentation, elles nécessitent un modèle qui représente le système et une description du comportement voulu et non voulu. Ceci est appelé une propriété. Selon les techniques, on utilise différentes manières de décrire les propriétés souhaitées du système.

Ces techniques peuvent être considérées comme non formelles, semi-formelles ou formelles, dépendamment de la technique et de la façon dont elle est faite. Les techniques formelles ont un modèle du système et une description des propriétés souhaitées qui sont écrits dans des langages formels. Les techniques non formelles ont un modèle et des propriétés définis dans des langages non formels. Finalement, les techniques semi-formelles sont un mélange des deux : le modèle est dans un langage non formel, mais les propriétés sont dans un langage formel. Par exemple, les techniques

[1] http://fr.wikipedia.org/wiki/Interpr%C3%A9tation_abstraite

empiriques sont à la base non formelles. Par contre, si l'on vérifie la présence d'une propriété formelle sur une trace d'exécution d'un simulateur ou d'un prototype, la vérification effectuée est semi-formelle. Cette technique particulière possède le nom de *Model-based testing* [11].

Une technique de vérification qui parait simple est la comparaison d'une implémentation à une autre dite de référence qui est sans erreur. Si pour toutes les entrées possibles, les résultats en sortie sont identiques, alors la nouvelle implémentation est valide. Ceci est utile si l'on crée une implémentation avec des contraintes différentes comme des optimisations pour l'espace, pour la vitesse ou pour le coût. Malheureusement, une version de référence n'est pas toujours disponible.

La simulation d'un modèle et la vérification manuelle du résultat est une technique qui ne fait pas appel à une implémentation de référence. Cette technique est possible pour de petits circuits, mais pour de plus gros, on doit automatiser cette vérification en donnant des séquences d'entrées à notre circuit et la liste des sorties correspondantes. Cette technique atteint vite sa limite. Premièrement, on doit énumérer toutes les séquences d'entrées possibles si on veut tout vérifier. Cette énumération peut être automatisée, mais on doit aussi associer la sortie valide attendue pour chaque entrée. Cette sortie peut être calculée, mais pour ce faire il faut une implémentation de la spécification. Un autre problème est qu'il est possible d'avoir des séquences d'entrées infinies même pour un système avec un nombre d'états fini. On a une telle séquence lorsque le système peut ne pas terminer. Il devient nécessaire de trouver un sous-ensemble de séquences d'entrées qui couvrent au mieux l'ensemble de toutes les entrées. Ceci peut être fait avec des classes d'équivalence, mais ce n'est pas facile à faire sans erreur et cela demande du temps.

Les techniques semi-formelles sont un mélange des techniques de vérification empirique et de vérification formelle. Elles prennent les propriétés à vérifier aux méthodes formelles et les vérifient sur des traces de simulation. Par contre, ces techniques ne règlent pas les problèmes pour trouver les traces de simulation à vérifier. C'est toujours au concepteur de les donner. Cette technique ne donne pas de preuves que le circuit respecte les propriétés dans tous les cas, sauf si on donne toutes les traces possibles d'entrée. Toutefois, faire la vérification exhaustive de cette manière est plus long que par les méthodes formelles, car cette technique vérifie plusieurs fois les mêmes états. Les techniques formelles ne vérifient qu'une fois chaque état des circuits. Par contre, elle est peut-être une étape de transition possible vers les techniques formelles qui ont besoin de propriétés pour faire la vérification.

On entend aussi parler de techniques par assertions. Ce ne sont pas des techniques de vérification, mais plutôt des techniques pour décrire les propriétés qui doivent être vraies. Ensuite, c'est au logiciel de valider les assertions (propriétés) par simulation (semi-formelle) ou par vérification formelle.

Les techniques formelles utilisent une autre méthode pour décrire ce qui est valide et ce qui ne l'est pas. Elles utilisent les propriétés. Une propriété est une description du comportement qui doit toujours être vraie. Par exemple, le comportement décrit peut être quelque chose qui ne doit jamais arriver (sécurité), qui doit arriver (vivacité) ou qui peut arriver. Par exemple, si l'on reçoit un signal X à vrai, le signal Y doit être vrai dans trois à cinq cycles plus tard et doit être faux avant. Les techniques de vérification formelle vérifient si cette propriété est toujours vraie. Si une propriété n'est pas vraie, plusieurs de ces techniques donnent un des contre-exemples ayant la plus courte longueur. C'est le cas des techniques formelles utilisées dans ce mémoire.

La vérification par simulation, formelle et semi-formelle fonctionne différemment durant les étapes de vérification. La simulation détecte un comportement non valide quand une sortie n'est pas valide. En vérification formelle et semi-formelle, c'est quand une propriété n'est pas valide. Lors d'une telle détection, la simulation et la vérification semi-formelle donnent un vecteur d'entrée de taille quelconque qui déclenche l'erreur. La vérification formelle donne un vecteur d'entrée de taille minimum, ce qui facilite la découverte de la cause du comportement non valide.

1.5. Objectifs

L'idéal serait d'avoir un système qui fait toute la vérification formelle automatiquement à toutes les étapes du développement d'un système; ce qui n'existe pas actuellement. L'objectif principal de ce mémoire est de développer une méthodologie pour vérifier de façon formelle une spécification existante. Dans ce contexte, on vérifiera la complétude interne, la cohérence et des propriétés additionnelles du concepteur.

Un des objectifs secondaires est de trouver un logiciel pour faire une étude de cas de la méthodologie créée avec un système connu.

La vérification de la complétude consiste à s'assurer que le système ne se bloque jamais faute de ne savoir que faire. En d'autres termes, pour tous les états que le système peut atteindre, la spécification doit préciser quelle action le système doit entreprendre à partir de cet état. En effet, la spécification

étant principalement textuelle, certains cas peuvent ne pas avoir été anticipés et risquent d'engendrer un comportement défectueux. Quant à la vérification de la cohérence, elle consiste à vérifier un ensemble de propriétés qui vérifient la cohérence du modèle par rapport à la méthodologie et à son comportement.

Pour vérifier une spécification textuelle, il importe d'en avoir une représentation formelle sur laquelle on peut raisonner. Dans le cadre de notre méthodologie, on a recours à un modèle en réseaux de Petri exprimant fidèlement le contenu de cette spécification textuelle sans rien omettre, ni surspécifier (ce que ferait une implémentation).

1.6. Contributions

La méthodologie, qui est présentée dans ce document, vérifie certaines propriétés, automatiquement ou non, au niveau de la modélisation du système. Dans un premier temps, on décrit la spécification textuelle par un modèle en réseaux de Petri. Cette description est réalisable en utilisant des outils tels que CPN Tools [12], GreatSPN [13] et CPN-AMI [14]. Ces outils permettent, entre autres, de faire la détection automatique d'états terminaux dans un réseau de Petri. On utilise cette analyse pour vérifier la complétude interne. En effet, l'absence d'états terminaux indésirables (correspondant à des interblocages engendrés par un problème de sous-spécification) est une des propriétés que le modèle devrait satisfaire pour garantir la complétude de la spécification.

Dans un deuxième temps, on doit prouver un ensemble de propriétés qui montre la cohérence de notre modèle. La vérification de ces propriétés est accomplie semi-automatiquement en utilisant le même modèle réseau de Petri. Ces propriétés sont vérifiables en examinant le graphe d'états que l'outil de réseaux de Petri peut générer automatiquement à partir du modèle. Afin de limiter l'explosion du nombre d'états de ce graphe, notre méthodologie proposera des techniques de modélisation et d'abstraction adéquates.

Dans un troisième temps, il est possible que le concepteur souhaite vérifier d'autres propriétés de la spécification. Dans les systèmes où le comportement est synchronisé à une horloge, les propriétés à vérifier sont majoritairement synchronisées à cette l'horloge. Pour les vérifier, on propose soit de dégager un sous-graphe du graphe d'états qui contient juste les états au front de l'horloge soit de modifier la propriété pour utiliser le graphe d'état complet. Il est alors possible de parcourir ce sous-

graphe pour vérifier les propriétés temporelles souhaitées.

Plus on avance dans le développement du modèle, plus il y a de détails. Ces détails supplémentaires finissent par causer une explosion combinatoire du nombre d'états qui entraîne une explosion du temps de calcul pour les techniques de model checking. C'est le problème principal de cette technique et on donne quelques moyens d'en minimiser l'impact.

Finalement, une étude de cas portant sur la spécification du bus AMBA AHB-Lite d'ARM est présentée pour valider notre méthodologie.

1.7. Plan du mémoire

Le chapitre 2 est un survol des différentes techniques de vérification formelle et des réseaux de Petri. Le chapitre 3 donne la méthodologie sans modèle de calcul. Le chapitre 4 explique la méthodologie avec le modèle de calcul des réseaux de Petri colorés et l'étude de cas est détaillée dans le chapitre 5. Le chapitre 6 conclut le mémoire et donne des pistes pour des travaux futurs.

2. Comparaison des techniques de vérification formelle

Dans ce chapitre, on présente les connaissances nécessaires pour comprendre la méthodologie présentée dans ce mémoire et ce qu'elle apporte.

2.1. Vérification formelle

Toutes les méthodes de vérification ont besoin de savoir ce qui est considéré comme valide ou non valide. En vérification formelle, on appelle *propriété* une caractéristique/comportement qui doit être présent(e) ou non. Il y a des propriétés de sûreté qui spécifient un comportement qui ne doit pas arriver et il y a des propriétés de vivacité qui spécifient qu'un comportement doit arriver tôt ou tard. Pour qu'un circuit fonctionne sans erreur, il doit respecter les propriétés de la spécification. C'est au concepteur de trouver les propriétés à partir de la spécification. C'est aussi à lui de s'assurer que l'ensemble de propriétés fournies est complet.

Il y a deux aspects à la vérification d'un système. Le premier est de vérifier que la spécification du système correspond aux besoins des utilisateurs. Et le deuxième est de vérifier que le système correspond à la spécification. Ici on ne traite que du dernier problème. La spécification étant sous forme textuelle, elle est sujette à diverses interprétations et cela peut amener à des erreurs lors de son implémentation. En fait, il est impossible de prouver formellement qu'une spécification sous forme textuelle est complète et cohérente. Le mieux que l'on puisse faire, c'est de trouver une liste la plus complète possible de propriétés à vérifier pour un système. Ensuite, on peut vérifier formellement cette liste de propriétés. Par contre, il n'existe pas à notre connaissance dans la littérature scientifique de moyen connu pour savoir si une liste de propriétés est complète ou non. C'est donc pour cela que c'est la responsabilité du concepteur de concevoir une *bonne* liste. Cela ne garantit pas que la liste soit complète.

2.1.1. Les propriétés

Les propriétés peuvent être vues comme une extension des assertions des langages de programmation. Une assertion en C vérifie qu'une propriété est valide à l'instant de son exécution dans le code et ne peut pas faire référence au futur ou au passé. Par contre, les propriétés peuvent

avoir une composante temporelle. Par exemple, on peut vérifier que si dans un état, une propriété A est vraie alors la propriété B est vraie dans le prochain état. Ceci pourrait correspondre à la réception d'un message et à l'envoi d'un accusé de réception au prochain cycle. Les propriétés A et B sont des propriétés atomiques; des propriétés de base qui utilisent seulement des opérateurs du langage de modélisation sur l'état courant.

Les propriétés qui doivent être vérifiées peuvent être écrites dans le même langage que le système. Par contre, cette approche est difficile à utiliser, car les propriétés ont souvent des formes particulières qui ne sont pas faciles à décrire dans les langages de description de matériel. C'est pourquoi il existe des langages spécialisés pour décrire ces propriétés.

Ces langages de description de propriétés sont basés sur une ou plusieurs logiques et on obtient ainsi des propriétés formelles[2]. Ces logiques sont divisées en deux catégories dépendamment de la manière qu'elle représente le temps : le temps linéaire et le temps arborescent. Le temps linéaire permet décrire des propriétés sur des séquences d'états. Le temps arborescent suppose qu'il y a plusieurs états successeurs possibles. La logique LTL (*Linear Temporal Logic*) et sa notation sont utilisées pour exprimer les propriétés de temps linéaire. Quant aux propriétés de temps arborescent, elles sont généralement exprimées dans la logique CTL (*Computational Tree Logic*).

Les propriétés exprimées en LTL ne permettent pas de tenir compte des différents branchements de l'exécution d'un système dans le temps. Ces branchements sont dus aux choix multiples possibles dans un état. La logique CTL permet de faire de la vérification de propriétés qui contiennent de l'information sur différentes branches d'exécution. Ces deux logiques ne sont pas équivalentes ni incluses l'une dans l'autre. La logique CTL* inclut ces deux logiques. Par contre, elle n'est pas souvent utilisée, car elle est plus difficile à vérifier formellement que CTL ou LTL. Du reste, il arrive souvent que CTL ou LTL soit suffisante pour exprimer les propriétés souhaitées. Par exemple, dans [15], on donne comme exemple que la propriété GFp n'a pas d'équivalent exprimable en CTL et que $AG(p \rightarrow ((ENq) \wedge (EN\neg q)))$ ne peut être traduite en LTL. Donc, aucune des deux logiques n'est un

[2] Dans la suite de ce mémoire, quand il y a une référence à des propriétés, ce sont des propriétés formelles

sous-ensemble de l'autre, bien qu'elles aient une sous-partie commune. La première propriété indique que peu importe dans quel état on est, dans le futur p doit obligatoirement arriver. La deuxième propriété indique que pour tous les chemins et quel que soit l'état où l'on se trouve, si la propriété atomique p est vraie dans cet état alors il existe au moins un prochain état où la propriété q est vraie et au moins un prochain état où q est fausse.

Les langages Property Specification Language (PSL) [16], System Verilog Assertion (SVA) [17] et Open Vera Assertion (OVA) [18] sont entre autres basés sur les logiques LTL ou CTL. Ils ont une syntaxe plus facile à utiliser et proposent parfois des extensions à ces logiques. Aussi, ces trois langages ont été conçus pour représenter des propriétés pour le matériel. Mais comme elles sont basées sur les logiques LTL ou CTL, on peut les utiliser pour modéliser des systèmes contenant du logiciel. Ceci correspond à notre domaine d'utilisation.

La logique CTL a été utilisée dans le langage Sugar [16] qui est devenu PSL. Cette logique s'avère toutefois trop difficile à utiliser pour la majorité des usagers. Le consortium qui a conçu PSL a rendu le sous-ensemble CTL optionnel et a ajouté la logique LTL comme logique principale. Il n'est donc pas nécessaire d'avoir une méthodologie qui supporte la vérification de propriétés CTL, car elle risque d'être peu utilisée. Du reste, LTL permet d'exprimer la majorité des propriétés nécessaires.

Trouver une liste de propriétés et les écrire dans un de ces langages n'est pas facile. Pour cette raison, il existe des bibliothèques de propriétés qui sont vendues par des compagnies ou qui sont disponibles gratuitement. Ces bibliothèques sont faites par des professionnels de la vérification et risquent peu de contenir des erreurs ou d'être incomplètes. L'utilisation de ces bibliothèques a pour conséquence d'accélérer le temps de vérification et ainsi de diminuer le coût de conception. Une de ces bibliothèques disponibles gratuitement est appelée Open Verification Library (OVL) [19].

2.1.2. Linear Temporal Logic (LTL)

La logique LTL [9, 10] est une logique qui permet de décrire des propriétés avec une composante de temps linaires. La sémantique de LTL est décrite avec une structure de Kripke qui est un quadruplet (S, I, R, Étiquette) où :

- S est un ensemble énumérable d'états

- $I \subseteq S$ est un ensemble d'états initiaux
- $R \subseteq S \times S$ est une relation de transition qui satisfait : $\forall s \in S.(\exists s' \in S \mid (s, s') \in R)$
- Étiquette : $S \rightarrow 2^{PA}$ est une fonction d'interprétation qui associe à chaque état S une valeur de vérité (vraie ou fausse) à chaque proposition atomique (PA)

Avec cette structure, on peut écrire des propositions. Ces propositions sont basées sur des propriétés atomiques. Une propriété atomique est une propriété de base qui ne contient pas d'opérateurs LTL. Par exemple, $x > 2$ est une propriété atomique. Il est à noter que tous les états ont un état successeur. On peut modéliser un état terminal en créant un état distinct qui boucle sur lui-même. Ensuite, on crée une transition des états terminaux vers cet état distinct.

Il y a deux opérateurs *standard* : $\neg$ (négation) et $\vee$ (disjonction, OU logique). Les deux autres sont des opérateurs temporels : N dénote l'opérateur « suivant » et $\bigcup$ dénote l'opérateur « tant que » (*until*).

Avec une proposition atomique p, les formules suivantes sont des formules LTL.

- p
- Si ϕ est une formule, alors $\neg\phi$ est une formule
- Si ϕ et φ sont des formules, alors $\phi \vee \varphi$ est une formule
- Si ϕ est une formule, alors $\mathrm{N}\phi$ est une formule
- Si ϕ et φ sont des formules, alors $\phi \bigcup \varphi$ est une formule
- Tout le reste n'est pas une formule LTL

Un chemin est une séquence de transitions dans un automate. Pour toutes formules ϕ dans le langage LTL, on définit une relation de satisfaction, $A, c \Rightarrow \phi$, sur un chemin c d'un automate A avec t qui représente une transition.

- $A, c \Rightarrow 1$
- $A, c \not\Rightarrow 0$

- $A,c \Rightarrow \phi \vee \varphi$ ssi $A,c \Rightarrow \phi$ ou $A,c \Rightarrow \varphi$
- $A,c \Rightarrow N\phi$ ssi $c = t \cdot c'$ et $A,c' \Rightarrow \phi$
- $A,c \Rightarrow \phi \bigcup \varphi$ ssi $A,c \Rightarrow \varphi$ ou $\exists n \mid c = t_1 ... t_n \cdot c'$ et $A,c' \Rightarrow \varphi$ et $\forall i \in \{1,...,n\}, A,t_1,...,t_n,c' \Rightarrow \phi$

Avec les opérateurs définis ci-haut, on peut ajouter d'autres opérateurs :

- $\phi \wedge \varphi \equiv \neg(\neg\phi \vee \neg\varphi)$
- $\phi \rightarrow \varphi \equiv \neg\phi \vee \varphi$
- $\phi \leftrightarrow \varphi \equiv (\phi \rightarrow \varphi) \wedge (\varphi \rightarrow \phi)$
- $true \equiv \phi \vee \neg\phi$
- $false \equiv \neg true$
- $F\phi \equiv true \bigcup \phi$,
- $G\phi \equiv \neg F \neg\phi$

Le symbole $\wedge$ est appelé la conjonction (l'opérateur logique ET), $\rightarrow$ est appelé une implication, $\leftrightarrow$ est appelé une double implication, *true* est appelé une tautologie, *false* est appelé une contradiction, F est appelé « fatalement » et G est appelé « globalement ».

On dit qu'une formule LTL est satisfaite pour une structure de Kripke (K) si elle est vraie pour tous les chemins qui commencent dans un des états initiaux : $K \Rightarrow \phi \; ssi \; \forall s \in I.(\forall \sigma \in Paths(s).\sigma \Rightarrow \phi)$.

2.1.3. Techniques de vérification formelle

Il existe deux catégories de techniques pour faire des preuves des propriétés : le model checking et la preuve par déduction logique (theorem proving).

Les techniques de model checking consistent à vérifier que les propriétés sont valides dans tous les états possibles du système. L'utilisateur définit le modèle et les propriétés à vérifier. La vérification des propriétés sur le modèle est complètement automatisée en passant par le graphe d'états.

Avec l'approche par déduction logique, il existe des systèmes qui permettent de faire des preuves automatiquement. Le problème est qu'ils ne sont pas encore capables de le faire sur des modèles industriels et ce n'est pas sûr que cela soit possible dans le futur. C'est pour cela qu'il existe des programmes d'assistance à la preuve, mais c'est à l'utilisateur de guider l'outil pour l'enchaînement des déductions. Ces logiciels vérifient automatiquement la preuve et peuvent même faire une partie de la preuve dans certains cas. La difficulté est que le niveau de compétence requis pour faire cette preuve est très élevé. Isabelle [8] et Higher Order Logic (HOL) [7] sont des exemples d'assistants de preuve.

Un des avantages des techniques formelles sur les autres techniques est que certaines d'entre elles donnent un contre-exemple le plus court possible. Ceci facilite grandement la découverte de la cause de l'échec de vérification, car il n'y a pas d'étapes inutiles dans le contre-exemple.

Le model checking présente une meilleure automatisation et est supporté par des outils plus conviviaux, on base notre méthodologie sur cette approche. Par contre, sa difficulté principale est l'explosion combinatoire du nombre d'états. Il faut que notre méthodologie prenne en compte cette difficulté, et propose des patrons de stratégies pour la limiter. L'explosion combinatoire du nombre d'états survient généralement quand le domaine des données est grand. Ceci arrive souvent quand il y a un bas niveau d'abstraction. Par exemple, dans un bus de donnée de 32 bits, il y a 2^{32} valeurs possibles sur le bus. Si on représente toutes les valeurs, cela crée une explosion combinatoire du nombre d'états, car il faut un état pour chaque valeur possible.

2.1.4. Modèles de calcul associés au model checking

On présente ici trois modèles de calculs pouvant être vérifiés par model checking : Communicating Sequential Process (CSP), les machines à états et les réseaux de Petri. Ces modèles de calcul sont la partie théorique/formelle de leur système de vérification. Les modèles de calculs ont un impact sur le langage de description de la spécification, sur les propriétés vérifiables et sur les techniques de vérifications des propriétés.

CSP est un modèle de calcul qui permet à des processus de communiquer par messages envoyés au travers de canaux de communication. C'est le seul à avoir des canaux de communications explicites dans le modèle de calcul. Par contre, cela peut-être simulés dans les autres langages.

Le deuxième modèle de calcul est les machines à états. Les machines à états finis sont des concepts qui sont bien connus. Donc, les outils qui les utilisent ont des langages qui sont rapides à apprendre, car les concepts utilisés sont connus.

Le troisième modèle est les réseaux de Petri. Les réseaux de Petri sont une famille de modèles de calcul. Ils ont tous un même héritage qui vient de la thèse de Carl Adam Petri en 1962. Pour la spécification de systèmes complexes, il existe diverses variantes de Petri de haut niveau comme les réseaux de Petri colorés (CPN), les réseaux bien formés (WN) et les réseaux de Petri stochastiques bien formés (SWN). Les réseaux de Petri ont une représentation graphique du modèle de calcul.

Les trois modèles de calcul permettent la description de systèmes synchrones et asynchrones. Un système synchrone est un système ou l'exécution est synchronisée à une ou plusieurs horloges. Donc, tous les calculs commencent à leur top d'horloge. Un système asynchrone fait le calcul quand les valeurs d'entrées arrivent ou changent. Un système est asynchrone entre des domaines d'horloges, car les exécutions dans chaque domaine ne sont pas exactement en même temps. Ceci correspond à notre domaine d'application, puisque les processus des systèmes qu'on considère possèdent zéro, une ou plusieurs horloges individuelles. Le transfert sécuritaire d'information d'une horloge à l'autre a été modélisé en réseau de Petri [20].

Les trois modèles de calcul ont en commun plusieurs points qui sont essentiels pour la description de spécification. En voici quelques-uns : non déterminisme des actions (plusieurs actions/réactions possibles), types de donnée complexes (booléens, scalaires, symboliques, tableaux), support de la hiérarchie et la réutilisation de modules, parallélisme et entrelacement des processus, communication entre processus par variables. En plus, les trois modèles de calcul ont des outils qui peuvent les simuler et qui donnent des contre-exemples avec la vérification formelle.

Ces trois modèles de calcul permettent de spécifier un système. La sélection entre ses trois modèles de calcul dépend des outils associés, des vérifications possibles et de la facilité d'utilisation du langage. Dans le cadre de ce mémoire, ce sont les réseaux de Petri qui ont été sélectionnés.

2.1.5. Langage et outils de modélisation pour le model checking

Il existe plusieurs langages dans lesquels le modèle à vérifier peut être décrit. Chaque langage est basé sur un modèle de calcul. Mais ils ne permettent pas toutes les mêmes vérifications. Par exemple,

l'outil SPIN et son langage Promela, acceptent les descriptions de systèmes synchrones et asynchrones, mais il ne permet pas de faire de l'analyse structurelle (voir section 2.2.7).

Les réseaux de Petri sont une famille de modèles de calcul et chacun a au moins un langage pour représenter le comportement d'un système. Plusieurs langages de propriétés ont été utilisés pour effectuer la vérification des réseaux de Petri, dont LTL et CTL. Les réseaux de Petri ont une représentation textuelle et graphique.

Si on compare les trois langages, Promela, la représentation graphique des réseaux de Petri et le langage de NuSMV, quant à leur facilité pour écrire le modèle de la spécification, on remarque que les réseaux de Petri sont les plus appropriés pour nos besoins. Promela n'est pas un très bon choix, car il ne permet qu'un type symbolique qui peut avoir au maximum 255 éléments. Cela est problématique, car avoir des types symboliques simplifie la description de notre modèle de haut niveau. En avoir plusieurs permet de séparer les différents concepts et ainsi d'éviter des erreurs. De plus, comme on est au niveau de la spécification, il y a de forte chance qu'elle contienne des symboles plutôt que des valeurs réelles. Par conséquent, avoir plusieurs symboles permet au modèle d'être plus près de la spécification plutôt que d'une implémentation de la spécification.

En plus, NuSMV et SPIN ne forcent pas de couvrir tous les cas possibles dans une exécution conditionnelle (ex. dans un « if ») : un cas par défaut est implicitement défini. Ceci limite la capacité de vérifier la complétude, car si on a une exigence manquante dans la spécification, on a un comportement qui est exécuté par défaut alors que rien n'est spécifié dans la spécification. Les réseaux de Petri obligent à mettre explicitement tous les cas d'une exécution conditionnelle, donc ils peuvent détecter plus d'erreurs.

Si on compare les vérifications possibles avec les outils SPIN, NuSMV et les réseaux de Petri, on remarque que les trois permettent minimalement de calculer le graphe d'états et de donner les états terminaux à partir de celui-ci. Comme les états terminaux permettent de vérifier la complétude interne, les trois outils permettent de le faire. Si on compare les langages existants des modèles de calcul quant à leur capacité de vérification, on voit que c'est encore les réseaux de Petri qui sont les plus appropriés pour notre approche. Car ils obligent à rendre explicite tous les cas d'une exécution conditionnelle et ils permettent de faire de l'analyse structurelle.

Tableau 1 : Comparaison des vérifications des outils SPIN, NuSMV et CPN Tools et ce qui est possible avec l'ensemble des outils de réseaux de Petri

Types de vérification\Outils	SPIN	NuSMV	CPN Tools	Réseaux de Petri
Invariant	Vérifie ceux donnés	Vérifie ceux donnés	Vérifie ceux donnés et en donne quelques-uns	Donne tous les invariants
BMC (Bounded Model Checking)	Non	Oui	Génération guidée du graphe d'états	Non
SAT (Satisfiability)	Non	Oui	Non	Non
Approximation de preuve	Oui	Avec BMC	Oui,	Pour le maximum ou minimum de jetons dans une place
BDD	Oui	Oui	Non	Oui
Réduction d'ordre partiel	Oui	Non	Non	Non
LTL, CTL et PSL	LTL	LTL, CTL, PSL*	ASK-CTL	LTL, CTL
Analyse structurelle	Non	Non, invariant?	Non	Oui
Vérification à la volée d'LTL	Oui	Non	Non	Oui

* Le sous-ensemble de PSL équivalent à LTL ou CTL. C'est juste la syntaxe qui change.

2.2. Introduction aux réseaux de Petri

Les réseaux de Petri ont été inventés par Carl Adam Petri en 1962 dans sa thèse de doctorat. Ce modèle de calcul a été développé par la suite par beaucoup de personnes. On se retrouve donc

aujourd'hui avec plusieurs catégories de réseaux de Petri. Elles ont toutes des bases communes qui datent de 1962. Une de ces bases qui aide à l'extraction des requis manquants est que l'espace d'états et les transitions entre ces états sont explicites. L'espace d'états est explicite avec les places. Les transitions entre les états sont explicites avec les transitions des réseaux de Petri.

2.2.1. Catégorie des réseaux de Petri

Il existe plusieurs types de réseaux de Petri. On en distingue trois niveaux [21]. Le premier niveau contient les réseaux de Petri qui n'utilisent que des valeurs booléennes comme jetons et où chaque place contient au plus un jeton. Le deuxième niveau contient les réseaux de Petri qui ont des jetons avec des valeurs entières et qui peuvent avoir plusieurs jetons par place. Ces places agissent comme des ensembles multiples. Le troisième niveau contient les réseaux de Petri de haut niveau dont les jetons peuvent avoir des données structurées. Plus le réseau de Petri est de haut niveau, plus il est facile de modéliser une spécification, car une meilleure abstraction est possible. Par contre, si le langage est trop puissant, son analyse formelle devient plus difficile.

En plus de ces catégories, il existe des caractéristiques dans certains réseaux de Petri qui simplifient la modélisation. Ils peuvent supporter une hiérarchie, des gardes (préconditions supplémentaires), des arcs inhibiteurs (si la condition est vraie, on désactive la transition), des PEPS (Premier Entré Premier Sortie) et des places fusionnées (places ayant leur jeton en commun). Il est possible de transformer un réseau de Petri contenant ces éléments vers un réseau de Petri équivalent qui n'en contient pas. Il y a aussi d'autres caractéristiques comme le choix stochastique de la transition à activer, l'utilisation de la logique floue, le support du temps dans les transitions et la possibilité d'avoir un niveau de priorité dans les transitions. La priorité des transitions peut être utilisée pour créer des transitions internes qui ne sont pas vérifiées. Ceci est utile pour diminuer le nombre d'états en éliminant les états entre les fronts de l'horloge.

Pour modéliser les spécifications d'un système électronique complexe, il est préférable d'utiliser un réseau de Petri de troisième niveau. Avoir un modèle de calcul de haut niveau facilite la modélisation de la spécification, car la différence de niveau de représentation est moindre qu'avec les autres catégories de réseaux de Petri. Il existe plusieurs réseaux de Petri de niveau trois dont les colorés (CPN, Colored Petri Net), les bien formés (WN, Well-formed Petri Net où WFPN, Well-Formed Petri Nets) et les stochastiques biens formés (SWN, Stochastic Well-formed Nets). Il est préférable que le

réseau de Petri supporte la hiérarchie et des arcs bidirectionnels pour simplifier la modélisation.

Il est à noter que les réseaux de Petri bien formés sont un sous-ensemble des réseaux de Petri colorés. Les concepteurs de ces réseaux ont gardé les aspects haut niveau qui sont faciles à vérifier formellement, mais ils ont retiré les aspects difficiles à vérifier formellement comme les fonctions SML.

2.2.2. Réseaux de Petri Place/Transition (P/T)

Les réseaux de Petri Place/Transition (P/T) sont les réseaux de Petri de base de plusieurs autres réseaux de Petri actuels. Ils sont présentés en premier pour aider à la compréhension des réseaux de Petri, car tous ces éléments se retrouvent dans le type de réseau choisi. Les réseaux P/T sont un bigraphe avec un 5-tuplets (S, T, F, M_0, W), ou

- S est un ensemble de place
- T est un ensemble de transition
- $F \subseteq (S \times T) \cup (T \times S)$ est un ensemble d'arcs de la forme $(S \times T)$ ou $(T \times S)$
- $M_0 : S \to \{0,1,2,...\}$ est une fonction qui donne la quantité de jetons initiaux à chaque place
- $W : F \to \{1,2,3,...\}$ est une fonction qui donne un poids à chaque arc

Les réseaux P/T sont constitués de places, de transitions, d'arcs et de jetons. Les trois premiers éléments forment la structure des réseaux de Petri. Une place peut contenir zéro, un ou plusieurs jetons sans ordre à un instant donné. C'est le nombre de jetons dans les places d'un réseau de Petri qui détermine un état. L'activation d'une transition change l'état du système en enlevant ou en ajoutant, selon le cas, des jetons dans les places. Pour qu'une transition soit activable, ses préconditions doivent être toutes vraies et on exécute ses postconditions lors de l'activation de la transition. Les pré et postconditions sont déterminées par les arcs. Les arcs de base sont orientés, unidirectionnels et toujours situés entre une place et une transition. Les deux types d'arcs sont les arcs entrants dans une transition (précondition) et les arcs sortants d'une transition (postcondition). Quand un arc va d'une place vers une transition, cette place est une place d'entrée de la transition et l'arc est un arc entrant. Si l'arc est de sens inverse, la place est appelée place de sortie et l'arc, arc sortant. Sur les arcs, il peut

y avoir une étiquette qui indique une quantité de jetons, s'il n'y en a pas on utilise la valeur 1. Lors de l'activation d'une transition, on ajoute ou on enlève des places la quantité de jetons indiquée sur l'étiquette. Lors de l'activation d'une transition, on enlève des jetons des places d'entrée de la transition. Ensuite, on ajoute des jetons dans les places de sortie de la transition activée. Il ne peut jamais y avoir une quantité négative de jetons dans une place.

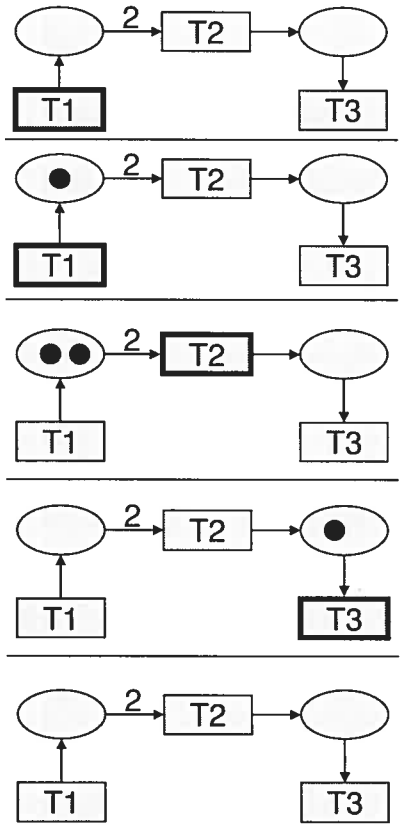

Figure 2: Exécution possible d'un réseau de Petri P/T

S'il n'y a pas assez de jetons dans la source d'un arc entrant, on ne peut pas exécuter cette transition. Initialement, les places ont un ensemble de jetons qui sont donnés par le concepteur. Les places et les transitions ont également des noms qui leur sont associés et doivent être uniques pour certains logiciels de vérification.

Dans certaines catégories de réseaux de Petri, il y a des arcs bidirectionnels. Ces arcs sont des pré et postconditions, mais ils n'enlèvent pas et n'ajoutent pas de jetons lors de l'activation de la transition. Ceci équivaut à avoir un arc d'une place vers une transition et un arc de cette transition vers cette

place qui a la même étiquette. Il est possible de transformer automatiquement un réseau de Petri avec des arcs bidirectionnels vers un réseau sans arc bidirectionnel.

Les réseaux de Petri ont une représentation graphique qui aide à la visualisation du modèle. Les symboles importants de la représentation graphique (Figure 2) sont les places (ellipses), les transitions (rectangles) et les arcs (flèches). Le nom des places et des transitions est à l'intérieur de leur symbole. Les étiquettes sont associées aux éléments pour leur ajouter de l'information nécessaire comme pour les arcs.

Prenons comme exemple le réseau de Petri de la Figure 2. À la première étape, il n'y a que la transition T1 qui est activable parce qu'elle n'a pas de précondition. À la deuxième étape, il y a un jeton dans la place de gauche, mais T2 n'est pas encore activable, car il n'y a pas assez de jetons dans la place. À la troisième étape, on peut activer les places T1 et T2. On choisit T2. Donc, on enlève 2 jetons de la place de gauche et on en ajoute un dans la place de droite. Maintenant, il y a T1 et T3 d'activables. On active T3 pour obtenir le même état que l'état initial. Ce réseau a un graphe d'état infini, car T1 est toujours activable puisqu'elle n'a pas de précondition.

Une transition est appelée morte si elle n'est jamais activable. Une transition vivante est une transition qui est activable dans toutes les composantes fortement connexes du graphe d'états (voir section 2.2.9). Donc, une telle transition peut être exécutée une infinité de fois dans les chemins infinis du graphe d'états (voir section 2.2.8).

On peut classer un modèle dans un réseau de Petri par le maximum de jetons qu'il peut y avoir dans une place. Si ce maximum est *k,* on dit que le réseau est *k-borné*. Pour les réseaux de Petri avec un maximum de 1 jeton par place, on dit qu'il est *1-borné* (*1 safe)*. C'est une propriété intéressante à avoir, car la théorie est plus développée pour ce genre de modèle, qui permet d'utiliser les meilleurs algorithmes de vérification existants.

2.2.3. Choix de l'outil de modélisation et de vérification

Pour répondre à nos besoins de modélisation et de vérification, on cherche un outil qui possède plusieurs propriétés. Tout d'abord, il faut qu'il supporte les réseaux de Petri de niveau trois pour simplifier la modélisation du système. Il faut aussi que l'outil soit capable d'effectuer les vérifications de complétude, de cohérence et des propriétés supplémentaires du modèle de façon formelle.

Les outils CPN Tools, GreatSPN et CPN-AMI supportent des réseaux de Petri à haut niveau. On a choisi CPN Tools pour notre étude de cas pour plusieurs raisons. Il est un des plus utilisés dans l'industrie. Il fait plusieurs analyses automatiquement du graphe d'état, il a une bonne interface utilisateur et une bonne liste de discussions pour obtenir des réponses à nos questions. Par contre, il ne fait pas d'optimisation automatique du graphe d'état, mais les auteurs de l'outil ont déjà publié [22] sur de tels algorithmes avec les réseaux de Petri colorés. Dans le futur, il sera possible que cela soit intégré dans l'outil. GreatSPN a une licence qui est plus difficile à obtenir, mais il a un modèle de calcul dont les transitions ont des priorités différentes et des transitions stochastiques. Ceci n'est pas nécessaire pour nos besoins.

Les réseaux de Petri colorés ont ce qui est nécessaire comme capacité d'analyse et comme capacité de représentation. En plus, ils sont utilisés dans l'industrie et dans l'académie à travers le monde. De plus, il y a plusieurs vérifications faites automatiquement après la génération du graphe d'états telles que le nombre maximal de jetons dans les places, l'ensemble multiple maximal des places, la détection des états terminaux, des transitions mortes et des transitions vivantes. Aussi, il est possible de générer le graphe de composantes fortement connexes (SCC) après le graphe d'état. Ceci permet de vérifier les propriétés d'équité (section 3.5). Enfin, des fonctions qui peuvent être utilisées pour naviguer directement dans le graphe d'états ou le graphe de composantes fortement connexes sont disponibles.

Une liste des outils disponibles utilisant les réseaux de Petri est donnée dans [23].

2.2.4. Réseaux de Petri colorés

Une version haut niveau des réseaux de Petri est les réseaux de Petri colorés. Le terme « couleur » est historique et synonyme d'un type de données. L'ajout principal comparé aux réseaux P/T est la couleur qui est associée aux jetons et aux places. Il y a aussi comme ajout l'utilisation de variables, de garde aux transitions et des fonctions dans le langage SML sur les arcs et dans les gardes. Ils sont définis formellement [24] comme un tuplet CPN = $(\Sigma, P, T, A, N, C, G, E, I)$

- Σ est un ensemble fini non vide de **type** de donnée (**couleur**)
- P est un ensemble fini de **places**

- T est un ensemble fini de **transitions**
- A est un ensemble fini d'**arcs**
- $N: A \rightarrow (P \times T) \cup (T \times P)$ est une fonction de **nœud**
- $C: P \rightarrow \Sigma$ est une fonction de **couleur**
- $G: \forall t \in T: [Type(G(t)) = B \wedge Type(Var(G(t))) \subseteq \Sigma]$ est une fonction de garde définie pour les transitions
- $E: \forall a \in A: [Type(E(a)) = C(p)_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$ est une fonction **expression d'arc** défini pour tous les arcs où p est une place de $N(a)$
- $I: \forall p \in P: [Type(I(p)) = C(p)_{MS}]$ est une fonction d'**initialisation** définie de P vers une expression fermée

Les réseaux de Petri coloré utilisent les ensembles multiples pour représenter les jetons dans les places et dans les arcs. Un tel ensemble est défini formellement par une paire (A, m) ou A est un ensemble et $m: A \rightarrow \mathbb{N}^+$ est une fonction de A vers l'ensemble $\mathbb{N}^+ = \{1,2,3,...\}$ des entiers positifs naturels. Pour chaque a dans A, la multiplicité (le nombre d'occurrences) de a est $m(a)$.

La représentation des ensembles multiples a la forme : *nb'jeton* [++ *nb'jeton*]* où [] indique que le contenu est optionnel et * qu'il peut être répété. Tous les jetons doivent être de la même couleur (type) dans un ensemble multiple. Dans une paire (*nb'jeton*), *nb* donne le nombre d'occurrences du jeton *jeton*. Par exemple dans l'ensemble *1'a ++ 2'v*, il y a 1 fois le jeton *a* et 2 fois le jeton *v*. L'opérateur ++ symbolise l'addition de paires (*nb'jeton*) dans un ensemble multiple.

Dans les réseaux de Petri colorés, chaque place contient un ensemble multiple de jetons initial et un ensemble multiple de jetons de l'état courant. La couleur des jetons dans une place et la couleur de cette place doivent être la même. Le concepteur peut créer ses propres couleurs. Ces dernières peuvent être un entier, une énumération, un sous-ensemble d'une couleur existante, une liste, un n-uplet (tuple, un « struct » en C), etc.

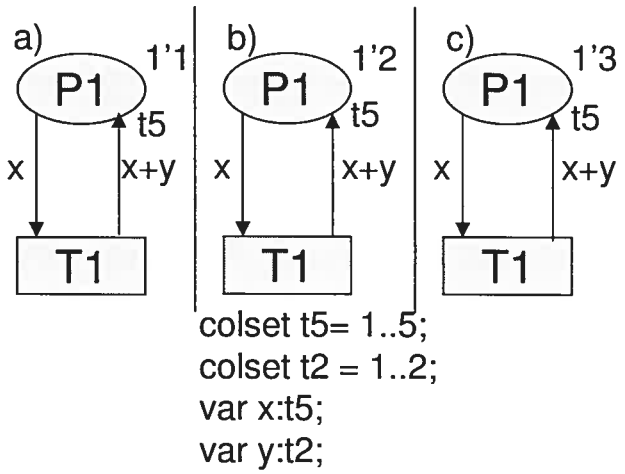

Figure 3 : Réseaux de Petri colorés où a) est le réseau initial et b) et c) sont les états suivants possibles de a)

Il y a deux raisons pour lesquelles les réseaux de Petri colorés sont non déterministes. La première est que la transition à activer est choisie aléatoirement quand il y en a plusieurs qui sont activables. La deuxième est qu'il y a la possibilité d'utiliser des variables sur les étiquettes des arcs et dans la garde des transitions.

Lorsqu'on vérifie si une transition est activable, il y a une étape de liaison des variables (*binding*). Cette étape détermine toutes les combinaisons de valeurs possibles pour les variables. S'il y a contradiction sur la valeur d'une variable, la transition est considérée comme non activable. Lors de la simulation, une des liaisons trouvées est choisie au hasard. Durant la vérification formelle, toutes les liaisons sont prises en compte. Les variables sur les arcs entrants prennent la valeur du jeton. S'il y a plusieurs jetons, la variable peut prendre toutes les valeurs des jetons comme liaison possible. Dans le cas où une variable n'est pas liée à des valeurs possibles, on prend toutes les valeurs possibles de sa couleur. Par exemple, dans la Figure 3, il y a deux liaisons possibles : x prend la valeur 1 et y prend la valeur 1 ou 2. Dans l'outil utilisé discuté plus loin, il faut aussi que la couleur ait au maximum 100 éléments. Sinon, une erreur est générée, car cela crée une très grosse explosion combinatoire.

Les transitions ont des gardes : des préconditions supplémentaires qui ne modifient pas le nombre de jetons dans les places. Pour que les transitions puissent être activées, il faut que les préconditions des arcs et de la garde soient vraies. Quand la transition est activée, des jetons dans les places sources des arcs entrants sont enlevés et dans les places destinations des arcs sortants, des jetons sont ajoutés.

2.2.5. Réseaux de Petri colorés hiérarchiques

Il existe une version hiérarchique des réseaux de Petri colorés [24]. Quand il y a une référence aux réseaux de Petri colorés, la version hiérarchique ou la version non hiérarchique peut être utilisé, car la version hiérarchique peut être transformée automatiquement vers la version sans hiérarchie. La hiérarchie est faite de plusieurs réseaux de Petri colorés qui sont appelés des pages : $(\Sigma_S, P_S, T_S, A_S, N_S, C_S, G_S, E_S, I_S)$. Comme les mots « place » et « page » commencent par la même lettre, les pages utilisent la lettre S comme symbole qui signifie sous réseau. Les éléments du réseau de Petri entier sont définis ainsi : $\Sigma = \bigcup_{s \in S} \Sigma_s$, $P = \bigcup_{s \in S} P_s$, $T = \bigcup_{s \in S} T_s$, $A = \bigcup_{s \in S} A_s$. Il est à noter que les sous réseaux partagent des éléments de Σ et ne partagent pas d'élément de T, P et A. Comme les éléments du réseau sont disjoints, on peut utiliser une fonction globale de couleur $C \in [P \to \Sigma]$. La fonction globale est définie à partir des fonctions locales : $\forall s \in S, \forall p \in P_s : C(p) = C_s(p)$. Par le même procédé, on peut définir des fonctions globales pour N, G, E, I. Les réseaux de Petri hiérarchiques sont définis formellement comme un tuplet $(S, SN, SA, PN, PT, PA, FS, FT, PP)$ tel que :

- S est un ensemble de pages tel que chaque page est un réseau de Petri coloré non hiérarchique : $(\Sigma_S, P_S, T_S, A_S, N_S, C_S, G_S, E_S, I_S)$. Les ensembles des éléments des réseaux sont disjoints : $\forall s_1, s_2 \in S : \left[s_1 \neq s_2 \Rightarrow \left(P_{s_1} \cup T_{s_1} \cup A_{s_1}\right) \cap \left(P_{s_2} \cup T_{s_2} \cup A_{s_2}\right) = \phi\right]$
- $SN \subseteq T$ est un ensemble de nœuds de substitution (transition de substitution)
- $SA : AN \to S$ est une fonction d'assignation de page telle qu'aucune page n'est une sous page d'elle-même :
  $\left\{s_0 s_1 \ldots s_n \in S^* \mid n \in \mathbb{N}^+ \wedge s_0 = s_n \wedge \forall k \in 1..n : s_k \in SA(SN_{s_{k-1}})\right\} = \phi$
- $PN \subseteq P$ est un ensemble de nœuds de port
- $PT : PN \to \{\text{in, out, i/o, general}\}$ est une fonction qui assigne le type des ports
- $PA : SN \to (p_1, p_2)$ est une fonction d'assignation des ports tels que :

- Les sockets et les ports sont reliés : $\forall t \in SN : PA(t) \subseteq X(t) \times PN_{SA(t)}$
- Les sockets sont d'un type correct :
  $\forall t \in SN, \forall (p_1, p_2) \in PA(t) : [PT(p_2) \neq general \Rightarrow ST(p_1, t) = PT(p_2)]$
- Les nœuds reliés ont la même couleur et des expressions d'initialisation équivalentes : $\forall t \in SN, \forall (p_1, p_2) \in PA(t) : [C(p_1) = C(p_2) \wedge I(p_1)\langle\ \rangle = I(p_2)\langle\ \rangle]$

- $FS \subseteq P_S$ est un ensemble fini d'ensembles de places fusionnées tels que les membres des ensembles fusionnés ont la même couleur et des expressions d'initialisation équivalente : $\forall fs \in FS : \forall p_1, p_2 \in fs : [C(p_1) = C(p_2) \wedge I(p_1)\langle\ \rangle = I(p_2)\langle\ \rangle]$
- $FT : FS \rightarrow \{\text{global, page, instance}\}$ est une fonction de fusion des types telle que les ensembles de pages et d'instances appartiennent à une seule page :
  $\forall \text{fs} \in \text{FS} : [FT(\text{fs}) \neq \text{global} \Rightarrow \exists s \in S : \text{fs} \subseteq P_S]$
- $PP \in S_{MS}$ est un ensemble multiple de pages primaires

Ce qui permet l'interaction entre les deux pages, ce sont des places qui sont fusionnées entre les deux pages. Ces places sont appelées *socket* dans la super-page et *port* dans la sous-page. La fusion de places est faite avec des ensembles de places qui partagent le même contenu (ensemble de deux éléments pour la hiérarchie). On peut voir ces places comme des alias entre elles. Donc, quand on modifie le contenu d'une des places fusionnées, on modifie le contenu des autres places fusionnées avec celle-ci. Utiliser des places fusionnées sert à la hiérarchie et à simplifier la modélisation ainsi que la représentation graphique lorsque cette dernière contient beaucoup d'éléments.

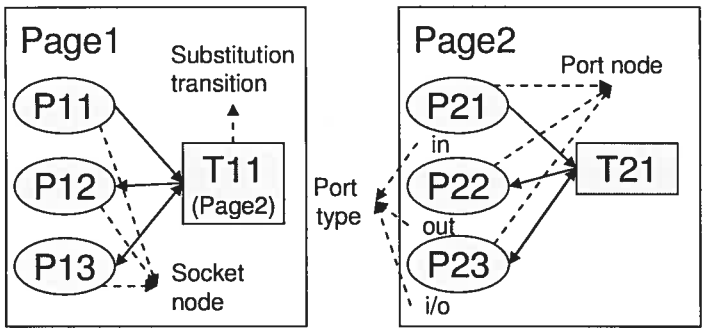

Figure 4 : Exemple de réseau de Petri coloré hiérarchique

Les transitions de substitution sont des transitions qui représentent une *sous-page* dans une *super-page*. Dans la Figure 4, il y a deux pages nommées *Page1* et *Page2*. La transition T11 est une transition de substitution qui représente la *Page2*. Les ports dans la sous-page ont un type (*in*, *out*, *i/o*) qui leur est associé pour limiter les interactions avec la super-page. Les places avec ces ports peuvent être vues respectivement comme des places d'entrée, de sortie, de paramètre et partagées de la sous-page. Il ne faut pas oublier qu'il faut mettre les arcs de manière à respecter le type des ports et qu'il faut associer les sockets et les ports. Il n'y a rien au niveau graphique qui indique cette liaison. Mais une convention de nommage claire permet au concepteur de s'y retrouver. Finalement, une page peut être instanciée une ou plusieurs fois dans d'autres pages, sinon elle est une page de niveau supérieur.

2.2.6. Exemple commun dans différents réseaux de Petri

Pour bien illustrer la modélisation en réseaux de Petri et montrer les différences entre ceux-ci, voici un exemple. Cet exemple est implémenté dans les réseaux de Petri Place/Transition, colorés et colorés hiérarchiques. Le système à représenter contient deux processus qui doivent prendre une ressource partagée pour travailler. Quand ils ont fini, ils remettent la ressource partagée à sa place en la modifiant.

La Figure 5a) montre le réseau de Petri Place/Transition de ce système. La place en haut est la place partagée. Le premier processus contient les transitions *debut1* et *fin1* ainsi que la place entre les deux. Pour le deuxième processus, ce sont les transitions *debut2* et *fin2*.

Initialement, les deux processus sont en compétition pour prendre la ressource partagée avec leur transition *debut1* et *debut2*. Lorsqu'un des processus la prend, il la met dans sa place interne. Comme les jetons n'ont pas de couleur, le processus termine en remettant le jeton sans modification dans la place initiale. Les Figure 5b) et c), sont les deux états atteignables à partir du premier état.

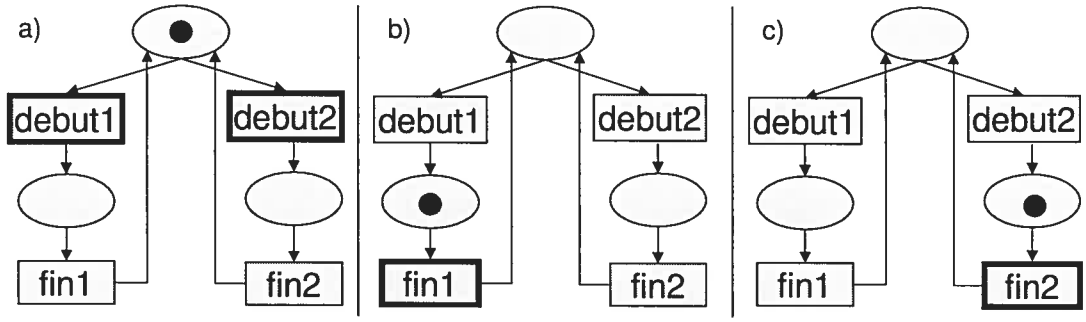

Figure 5 : Exemple commun en réseaux de Petri Place/Transition

La Figure 6 représente le même système en réseaux de Petri colorés. Il montre l'état initial ainsi que les deux états suivants possibles. La différence qu'il y a avec le réseau de Petri Place/Transition est qu'il utilise la couleur *BOOL*, `colset BOOL = bool`; ainsi que les variables *b* et *nb*, `var b, nb : BOOL;`. De plus, les places ont les noms suivants : *tmp1*, *tmp2* et *partage*. La place *partage* est initialisée avec un jeton de valeur *true*. L'avantage de l'utilisation des couleurs est que le processus termine en remettant un jeton avec une valeur aléatoire dans la place partagée. Cette modification est faite par la variable *nb* qui n'est liée à aucune valeur, donc c'est le simulateur qui choisit la valeur.

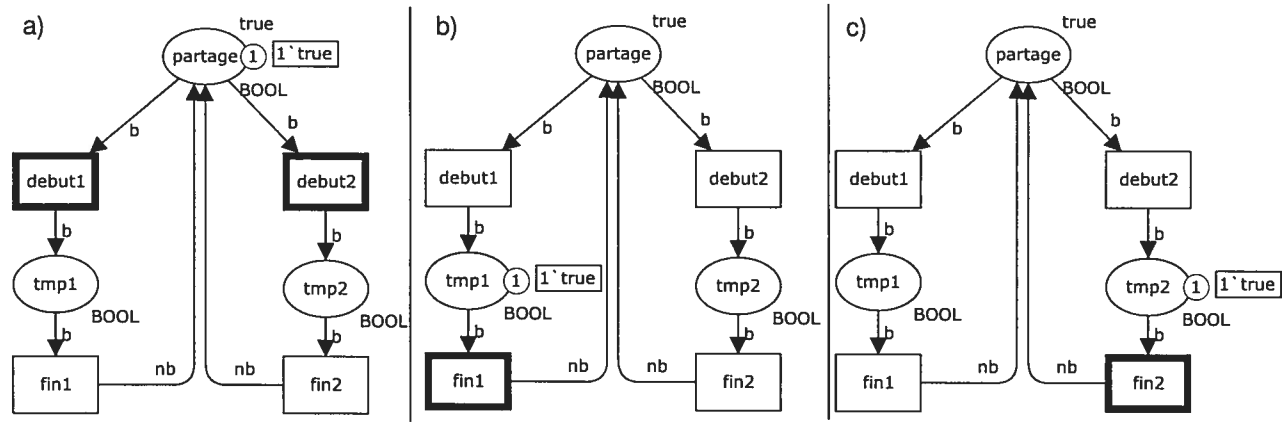

Figure 6 : Exemple commun en réseaux de Petri colorés

La dernière version du système est en réseaux de Petri colorés hiérarchiques. Il est représenté dans

les Figure 7 et Figure 8. Ce nouveau réseau a exactement le même comportement que la version en réseaux de Petri colorés, mais il est plus facile à créer et il garantit que les deux processus ont le même comportement. Ainsi lorsqu'on doit modifier les processus, on ne risque pas d'en oublier un.

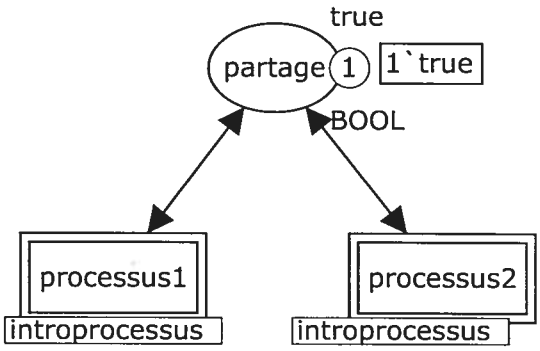

Figure 7 : Page *supérieure* d'un exemple en réseau de Petri colorés hiérarchiques

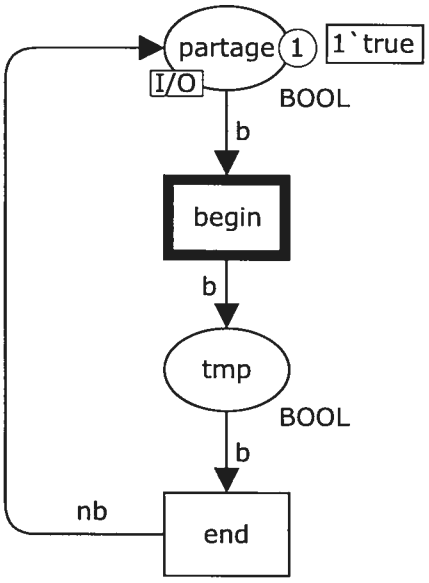

Figure 8 : Page *introprocessus* d'un exemple en réseaux de Petri colorés hiérarchiques

2.2.7. Analyses possibles des réseaux de Petri colorés

Les réseaux de Petri colorés permettent plusieurs types d'analyse. Celles utilisées dans notre méthodologie sont les suivantes :

- l'analyse de propriétés sur le graphe d'états

- l'analyse de propriétés sur le graphe de composantes fortement connexes
- l'analyse de propriétés par la structure du modèle

Les graphes des deux premières analyses sont expliqués dans les sous-sections suivantes. L'analyse des graphes est faite soit manuellement ou par des algorithmes de vérifications qui permettent de vérifier des propriétés sur les graphes. L'analyse de la structure du modèle est de prendre le modèle et de vérifier directement des propriétés sur la structure du modèle. Souvent, cela donne des propriétés qui sont plus faibles que par l'analyse des graphes, mais c'est plus rapide.

Il y a au moins un autre type d'analyse possible non utilisé dans notre méthodologie et qui se nomme l'analyse de performances [25, 26] qui utilise la simulation et donc n'est pas formelle.

On utilise la vérification de propriétés pour vérifier la complétude, la cohérence et pour permettre au concepteur de vérifier d'autres propriétés. Les vérifications utilisées dans notre méthodologie sont décrites plus loin. L'outil choisi, CPN Tools, génère automatiquement un rapport avec plusieurs propriétés formelles sur le graphe d'états et sur le graphe de composantes fortement connexes.

2.2.8. Graphe d'états

Il est possible de générer automatiquement le graphe d'états pour les réseaux de Petri si celui-ci est fini. C'est un graphe où chaque nœud n et m sont des états. Les arcs orientés (n, m) entre les noeuds indiquent les transitions possibles entre les états. Les arcs sortants d'un état correspondent à toutes les liaisons possibles des variables des transitions activables dans cet état. Si dans un état, il y a x transitions activables avec un total de y liaisons différentes, cet état va avoir y arcs sortants.

Si l'on prend le réseau de Petri de la Figure 2avec une limite de 2 jetons par place, on obtient le graphe d'états de la Figure 9. L'état est représenté comme une paire (a, b) où a est le nombre de jetons dans la place de gauche et b est le nombre de jetons dans la place de droite. L'état initial est (0,0). Les états de la trace d'exécution du réseau de Petri sont : (0, 0), (1,0), (2,0), (0,1), (0,0).

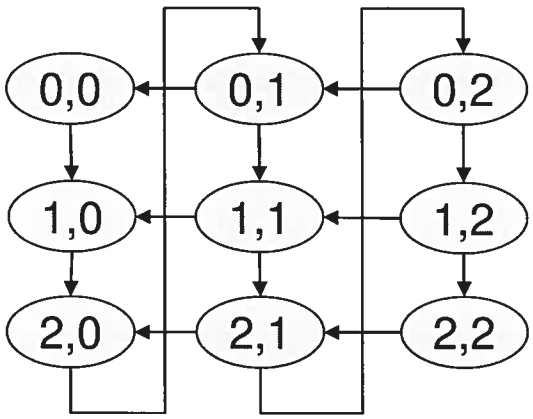

Figure 9 : Graphe d'états de la Erreur ! Source du renvoi introuvable.

La Figure 10 est le graphe d'états du réseau de Petri coloré de la Figure 3. L'état est décrit par l'ensemble multiple de la seule place du réseau. Il y a toujours un seul jeton dans cette place et initialement son jeton a une valeur de 1. Dans l'état initial, il y a seulement une transition activable, mais elle a deux liaisons possibles : la variable x peut prendre seulement la valeur 1 et la variable y peut prendre la valeur 1 ou 2. Donc, il y a seulement deux choix d'état suivant : soit le jeton de la place a la valeur 2 ou la valeur 3. Pour terminer la génération du graphe d'états, on trouve les états suivants de ces nouveaux états comme on vient de le faire.

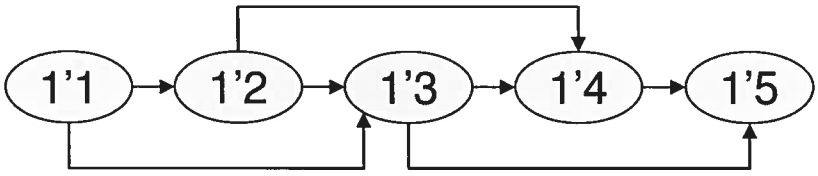

Figure 10 : Graphe d'états de la Figure 3

On peut remarquer que la place *1'5* n'a pas d'état suivant. Un tel état est appelé un état terminal.

2.2.9. Graphe de composantes fortement connexes

Le graphe de composantes fortement connexes a comme nœud une composante fortement connexe maximale du graphe d'états. Une composante fortement connexe est un ensemble de nœuds d'un graphe avec la propriété qu'à partir de tous les nœuds de l'ensemble on peut atteindre tous les autres nœuds. En d'autres mots, dans un tel type de composante, il est possible d'aller de n'importe quel état vers n'importe quel autre état en passant, si nécessaire, par d'autres états si tous ces états sont dans la

composante. Un ensemble d'états cyclique est une composante fortement connexe, mais l'inverse n'est pas toujours vrai.

Une composante fortement connexe maximale est une telle composante avec la condition supplémentaire qu'il est impossible d'en trouver une plus grande qui inclut cette composante. Toutes les composantes fortement connexes maximales d'un graphe sont disjointes et tous les états de ce graphe sont dans une de ces composantes. Il se peut que certaines de ces composantes maximales du graphe initial contiennent un seul nœud. Dans ce cas, cette composante maximale est qualifiée de triviale.

Par exemple, dans le haut de la Figure 11, on a un graphe d'état avec deux composantes fortement connexes maximales, donc le graphe de composantes fortement connexes a deux nœuds. Les arcs entre les nœuds de ce graphe sont les arcs entre les nœuds du graphe d'états qui vont d'une composante fortement connexe maximale à un autre. Pour terminer cet exemple, on peut dire que le nœud *2'* est un nœud trivial.

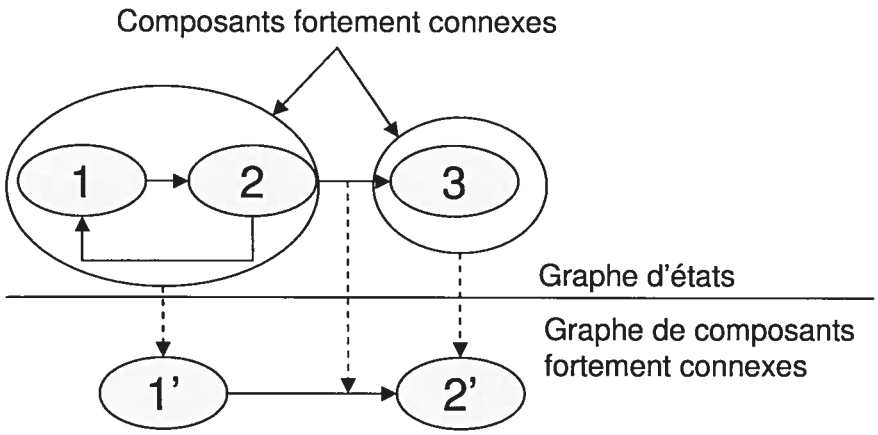

Figure 11 : Transformation d'un graphe d'états vers un graphe de composantes fortement connexes

Si un graphe de composantes fortement connexes a des nœuds qui représentent plus d'un état du graphe d'états correspondant, il a pour propriété d'être plus petit et de contenir moins d'information que celui-ci. Comme il est plus petit, il peut être utilisé pour vérifier des propriétés plus rapidement si l'information enlevée du graphe d'états n'est pas nécessaire pour ces propriétés. Il est intéressant de passer par ce graphe seulement si l'on gagne plus de temps pour vérifier les propriétés que le temps pour générer ce graphe. C'est le cas dans la méthodologie proposée, car ce graphe est utilisé pour

vérifier plusieurs propriétés. On récupère ainsi le temps mis pour le générer. La génération de ce graphe est de l'ordre de *max(|Arc|,|Noeud|)* du graphe d'états.

Le graphe de composantes fortement connexes de la Figure 10 est le même que le graphe d'états, car il n'y a pas de composantes fortement connexes non triviales. En effet, il n'y a pas de cycle et il est impossible d'aller vers un état où on a déjà été. En plus, on peut dire que tous les états de ce graphe sont triviaux, car ils correspondent à un seul état dans le graphe d'états. Pour ce qui est de la Figure 9, le graphe de composantes fortement connexes est d'un seul état, car il est possible d'aller de n'importe quel état vers n'importe quel état avec des états intermédiaires.

2.2.10. Avantages/inconvénients des réseaux de Petri colorés

Il y a plusieurs avantages à l'utilisation des réseaux de Petri comme modèle de calcul. Ils ne sont pas nouveaux et sont très étudiés partout dans le monde. Il y a beaucoup de recherches qui ont été faites et qui sont faites sur ces réseaux. La version colorée, comme tous les réseaux de Petri, permet une vérification formelle des propriétés et des requis manquants. La force des réseaux de Petri colorés, c'est qu'ils permettent une description plus facile des spécifications.

Par contre, les réseaux de Petri présentent plusieurs inconvénients. Les principaux sont reliés au fait que les outils existants utilisent souvent des versions différentes de réseaux de Petri et des formats de description différents. Ceci rend difficilement réutilisables les outils existants actuellement. Il y a en développement un standard proposé et qui se nomme : *Petri Net Markup Language (PNML)* [27, 28]. Ce standard a pour but de rendre communs les formats de description des modèles de réseaux de Petri. Ainsi, il sera plus facile dans un avenir proche de réutiliser les outils développés. Il ne restera plus qu'à faire des transformations de modèle lorsque cela sera utile.

Une introduction aux réseaux de Petri colorés est donnée dans des articles de Kurt Jensen [29-31]. Une introduction à la théorie est présentée dans l'article [32]. Dans les livres [24, 33, 34] faits par le concepteur du logiciel CPN Tools [12], tous les aspects des réseaux de Petri colorés sont décrits. Pour la vérification de systèmes électroniques au niveau matériel, il y a plusieurs articles qui incluent une description de ses différents éléments constituants. La vérification d'un arbitre en cascade est décrite dans l'article [35], la vérification d'un processeur d'architecture super scalaire dans [36] et la vérification d'un processeur VLSI dans [37].

Les réseaux de Petri colorés peuvent utiliser toute la puissance de SML pour faire des fonctions dans les arcs, les gardes, etc. On ne discutera pas de la programmation en SML dans ce document, car cela inutile, puisque la méthodologie n'en a pas besoin. Par contre, l'étude de cas les utilise pour simplifier le modèle. La méthodologie utilise principalement des éléments qui sont communs avec les réseaux de Petri bien formés.

2.3. Flot proposé

Le flot de conception proposé ne subit pas de modification. Ce qui change c'est qu'on fait de la vérification formelle au début du flot de conception pendant la modélisation comportementale. Ceci permet de vérifier au début de la conception la complétude de la spécification et la cohérence du modèle. Actuellement, cette modélisation est faite principalement dans des langages qui permettent la simulation. On propose d'utiliser des langages de modélisation qui permettent de faire de la vérification formelle par model checking. On utilise les réseaux de Petri dans la méthodologie proposée.

2.4. Complétude de la spécification

Il y a deux types de vérification de complétude : la vérification de complétude interne et la vérification de complétude externe. La complétude interne est la complétude par rapport aux éléments déjà dans la spécification. On vérifie l'agencement des éléments de la spécification pour vérifier s'il n'existe pas d'agencement pour lequel aucune action n'est définie.

La vérification de la complétude externe vérifie que rien d'important n'a été oublié dans la spécification.

La limite de la vérification de la complétude interne est qu'elle ne permet pas de trouver tous les requis manquants. Par exemple, si la spécification est vide, elle ne trouve rien de manquant. Elle est par contre utile pour vérifier qu'il n'y a pas de combinaison oubliée dans la spécification ou pour vérifier l'ajout/retrait de requis dans la spécification. Ceci implique que le domaine d'entrées des fonctionnalités soit bien défini.

La méthodologie proposée vérifie la complétude interne et permet ainsi de déterminer si la spécification contient des requis manquants. On ne s'intéresse pas ici à la complétude externe qui est

un problème indécidable en général. Seules les approches méthodologiques de saisie des besoins peuvent contribuer à l'améliorer sans pouvoir le garantir.

2.5. Vérification de cohérence

Il y a plusieurs types possibles de cohérence. Premièrement, il y a la cohérence syntaxique et sémantique. La cohérence syntaxique vérifie que le modèle construit respecte le langage dans lequel il a été écrit. Dans ce mémoire, il est supposé que les outils utilisés vérifient la cohérence syntaxique.

La cohérence sémantique est ici définie comme un ensemble de propriétés qui vérifie que le modèle est bien formé, c'est-à-dire qu'il respecte sa logique de modélisation. Dans ce mémoire, des propriétés qui peuvent être utilisées pour vérifier cette cohérence, qui sont indépendantes du modèle de calcul, sont présentées dans le chapitre 3. Dans le chapitre 4, ce sont des propriétés qui vérifient la cohérence, mais qui sont dépendantes des réseaux de Petri et de la méthode utilisée pour représenter la spécification sont présentées.

Ces propriétés sont vérifiées semi-automatiquement. Il faut que le concepteur vérifie l'information obtenue par l'analyse automatique du modèle. Si le modèle est cohérent, on peut déduire que la spécification est cohérente si le modèle lui est fidèle.

2.6. Comparaison de notre approche à celles existantes

Il est intéressant de comparer l'approche proposée dans ce mémoire (basée sur les réseaux de Petri) avec une approche basée sur SystemC [38] ou ESys.Net [39]. SystemC et ESys.Net permettent de modéliser un système à haut niveau comme notre méthodologie. SystemC est un simulateur utilisant C++ qui permet de décrire le comportement d'un circuit. Il a pour but de simuler le comportement à haut niveau de façon à faciliter le choix de l'architecture. L'avantage de simuler à haut niveau est que la simulation est plus rapide. Ceci permet de simuler plus de variantes de l'architecture et de choisir la meilleure.

ESys.Net a été conçu à l'Université de Montréal pour combler certaines lacunes de SystemC. La première version d'ESys.Net est donc très semblable à celle-ci. La plus grosse différence concerne l'introspection du modèle qui est facilement accessible en séparant le simulateur du modèle et l'ajout de points d'encrage dans le simulateur. L'introspection est la capacité de trouver statiquement ou

dynamiquement de l'information sur le modèle compilé (ex : le nombre de modules, leurs variables et fonctions internes, etc.). Ceci permet de créer beaucoup plus facilement des outils complémentaires comme des vérificateurs [40].

Notre approche propose de faire la vérification formelle de propriétés telles la complétude et la cohérence. Par contre, il y a une condition : que le modèle ait un nombre fini d'états. Ceci implique que le graphe d'états est fini et ses composantes peuvent donc être énumérées. Si ce n'est pas le cas, notre approche propose d'étudier le modèle par simulation, comme en SystemC ou avec ESys.Net.

Les trois approches, SystemC, ESys.Net et celle proposée, permettent de faire de l'analyse de performance, de faire de la vérification de propriétés par observation. Par contre, seule notre approche est conçue pour permettre de faire de la vérification formelle si le modèle possède la propriété énoncée précédemment. En ce moment, seul le développement en SystemC peut être synthétisé sans le réécrire dans un autre langage, par un logiciel commercial [41]. Il y a déjà eu de la recherche dans cette direction pour les réseaux de Petri [42]. Il ne faut pas oublier que seul SystemC est difficile d'introspection pour des outils externes. Ceci rend son extension difficile.

On peut voir que la différence principale est la possibilité de faire de la vérification formelle et l'énumération des requis manquants. En plus, cette vérification formelle est plus facile, car les réseaux de Petri sont plus abstraits qu'un langage de programmation.

2.6.1. Comparaison avec RAVE

RAVE est un système de vérification formelle de spécifications [43]. Les réseaux de Petri colorés sont un meilleur choix que RAVE pour plusieurs raisons.

Premièrement, on a essayé de modéliser notre étude de cas dans RAVE et cela s'est avéré trop difficile, car le langage que ce dernier utilise est de trop bas niveau (similaire aux réseaux de Petri Place/Transition).

Deuxièmement, RAVE vérifie la complétude en donnant tous les états terminaux, même ceux qui ne sont pas atteignables. Un état terminal est atteignable si on peut s'y rendre dans l'espace d'états à partir des états initiaux. S'il n'est pas atteignable, c'est un cas qui n'arrivera pas. Donc avec RAVE, le concepteur doit trier ces états terminaux pour savoir lesquels sont atteignables. Avec les réseaux de Petri, on trouve seulement les états terminaux atteignables.

Troisièmement, le dernier avantage des réseaux de Petri est que RAVE est supporté par un seul outil et qu'un seul petit groupe y a travaillé. Il reste donc plus de développements à faire sur les outils et la recherche de RAVE que si l'on prend les réseaux de Petri.

RAVE a quand même un avantage si on le compare aux réseaux de Petri. L'avantage est qu'il peut vérifier des propriétés LTL facilement. Pour ce faire, un traducteur du langage de RAVE vers le langage Promela de l'outil SPIN a été créé en quelques jours. Ensuite, on peut utiliser SPIN pour vérifier les propriétés LTL. Cette traduction vers Promela à partir des réseaux de Petri colorés n'est pas possible. Il faut donc créer un nouvel outil qui vérifiera les propriétés LTL à partir des réseaux de Petri.

2.7. Justification de la méthodologie

Les méthodes formelles basées sur le model checking ont l'avantage de faire la preuve automatiquement des propriétés dans tous les états si le modèle est fini. La preuve est faite à partir du modèle et des propriétés souhaitées du modèle. Et si une propriété est fausse, les algorithmes utilisés dans ce mémoire donnent un contre-exemple parmi les plus courts possible.

Pour la modélisation, on a choisi les réseaux de Petri colorés, car c'est un modèle de calcul de haut niveau et qu'il permet de modéliser le type de spécification voulu. Le fait que le modèle soit de haut niveau aide à la modélisation, car cela évite de descendre dans des détails pour vérifier la spécification. On peut représenter nos spécifications en réseau de Petri, car ils permettent de modéliser des systèmes synchrones et asynchrones.

2.8. Problème d'explosion combinatoire

Plusieurs vérifications formelles sont faites à partir du graphe d'états du modèle. Un graphe d'états est un graphe dirigé dont les nœuds sont les états et les arcs dirigés entre les nœuds sont les transitions possibles entre les états. Le graphe d'états a un état ou un ensemble d'états de départ. On dit qu'un état est atteignable s'il existe une séquence de transitions pour l'atteindre à partir de l'état initial du modèle. Tous les états dans le graphe d'états sont atteignables. Les états qui ne sont pas atteignables ne sont pas dans le graphe.

Les techniques de model checking sont sujettes à un problème d'explosion combinatoire du nombre d'états. Une explosion combinatoire du nombre d'états est une très grande augmentation du nombre d'état. L'augmentation du nombre d'état doit être plus que linéaire pour les paramètres modifiés. Par exemple, un protocole de transfert présente une explosion combinatoire du nombre d'états relative à la taille m du bus : il faut considérer les 2^m valeurs transmissibles sur ce bus de m bits. Ceci cause des problèmes, car si le bus est construit sur 32 bits, il y a 2^{32} valeurs transmissibles et chaque valeur crée au moins 1 état. Souvent, on peut diminuer ou éliminer l'explosion combinatoire en employant certaines techniques d'abstraction des données.

Cela dit, il n'y a pas de solution miracle au problème d'explosion combinatoire du nombre d'états. Par contre, il existe plusieurs techniques qui pallient ce problème. Ces techniques peuvent être automatiques, semi-automatiques ou manuelles et sont séparées en deux catégories : celles qui diminuent le nombre d'états et celles qui évitent le problème en ne générant pas le graphe d'état. Certaines propriétés peuvent être vérifiées sans le graphe d'états, par exemple certaines utilisent la structure du modèle. Pour diminuer le nombre d'états, on peut abstraire certaines informations du modèle pour diminuer le nombre d'états. Comme dans l'exemple du protocole de transfert, on peut représenter les valeurs transmissibles possibles par un symbole. Ce symbole représentant n'importe quelle valeur possible. Ceci diminue le nombre de valeurs possibles dans le modèle et diminue en conséquence le nombre d'états. Malheureusement, ceci n'est pas toujours possible, car cela fait perdre de l'information sur le système vérifié et peut dissimuler des erreurs. Dans notre exemple du bus de 32 bits, on peut créer deux valeurs pour le bus, une pour dire qu'il y a une donnée et une pour dire qu'il n'y a pas de donnée. S'il n'y a aucun traitement qui dépend de la valeur transférée, cette abstraction ne modifie pas les résultats de la vérification. Il existe d'autres techniques de modélisation qui diminuent l'explosion combinatoire d'un modèle.

Le chapitre 3 décrit quelques unes de ces techniques, indépendamment de leur implémentation.

3. Méthodologie de haut niveau

Ce chapitre, décrit la méthodologie proposée indépendamment du modèle de calcul, du langage et de l'implémentation.

3.1. Étape de la modélisation et de la vérification

La méthodologie pose certaines hypothèses. Si elles ne sont pas remplies, c'est au concepteur de faire le nécessaire pour que les hypothèses soient valides. Une des hypothèses est que la spécification est séparable en modules. Une seconde hypothèse veut que la spécification donne une machine à états finis ou qu'il soit possible d'en obtenir une. Cette supposition n'est pas obligatoire, mais si elle n'est pas remplie, c'est au concepteur de trouver le moyen de modéliser chaque module dans le langage dans lequel le modèle est décrit. Une troisième supposition est qu'il est possible de faire un système avec un nombre fini d'états. On ne traite pas les cas ayant une infinité d'états.

La méthodologie proposée (Figure 12) commence par une spécification textuelle. Elle est séparée en deux branches principales : une branche de modélisation avec vérification de complétude interne et de cohérence et une branche optionnelle de vérification avancée.

La première branche consiste à modéliser l'interconnexion et l'interface des modules. Une première fonctionnalité de la spécification est ajoutée, ce qui permet de vérifier la finitude, la complétude, la cohérence et l'explosion combinatoire. Généralement, ce premier modèle est à un haut niveau d'abstraction. On obtient par raffinement un modèle de plus en plus détaillé et proche d'une réalisation finale. Les vérifications peuvent aussi être faites sur ces modèles raffinés. On peut aussi détecter les explosions combinatoires de l'espace d'états et modifier le modèle en conséquence.

On peut vérifier que l'espace d'états est fini. Il y a une discussion sur ce sujet, même si la méthodologie à comme hypothèse que le modèle a un espace d'états fini, car plusieurs algorithmes de vérification ne terminent pas si le système possède une infinité d'états.

Dans la deuxième branche de la méthodologie sur la vérification avancée, les propriétés qui doivent être respectées par le modèle qu'on construit sont ressorties de la spécification. Ces propriétés peuvent être de tout type, car la vérification est faite au niveau du graphe d'états accessibles. Pour

simplifier, on décrit comment vérifier les propriétés des modules synchrones seulement au coup d'horloge et les propriétés des modules asynchrones à chaque calcul effectué. Les propriétés ainsi dégagées de la spécification pourront être réutilisées pour vérifier les implémentations de la spécification. C'est au concepteur de vérifier que la liste de propriétés est complète.

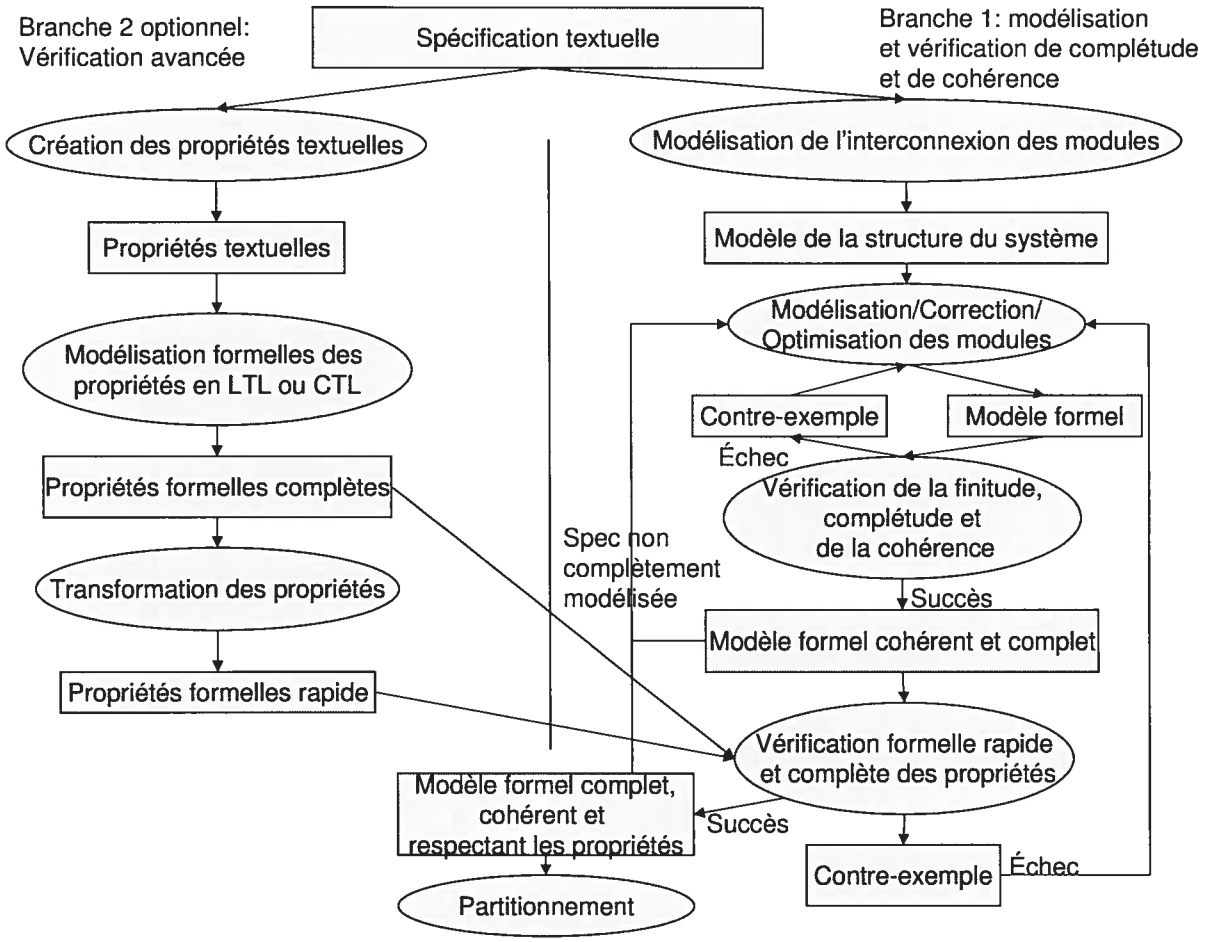

Figure 12 : Méthodologie proposée

3.2. Type de vérification

La méthodologie étant basée sur le model checking, on fait la vérification principalement avec le graphe d'états. Pour plusieurs propriétés, il est possible de les vérifier avec le graphe de composantes fortement connexes au lieu du graphe d'états (section 4.3.4).

Pour certaines propriétés, on peut diminuer davantage le temps de vérification en utilisant la

structure du modèle. Ces optimisations sont dépendantes du modèle de calcul. Donc, ces optimisations sont expliquées dans le chapitre 4 où on utilise cette méthodologie avec les réseaux de Petri.

3.3. Vérification de la finitude de l'espace d'état

Notre méthodologie a comme hypothèse que le graphe d'états est fini. Il est quand même utile de vérifier cette propriété, car plusieurs algorithmes de vérification n'arrêtent pas si le graphe est infini.

Si le modèle ne possède que des types de donnée finis et un nombre maximal fini de variables, le modèle est fini. Certains modèles de calcul, comme les FSM, ne permettent que ce type de modèle, donc ils ont toujours un graphe d'états fini. Pour les modèles de calculs pour lesquels ce n'est pas le cas, il peut y avoir des techniques qui sont dépendantes du modèle de calcul pour vérifier la finitude du graphe d'état. La technique pour vérifier si le graphe d'états est fini pour les réseaux de Petri est décrite dans le chapitre 4.

3.4. Vérification de la complétude

Une fois la vérification de finitude du modèle terminée, on peut vérifier la complétude interne. Cette vérification se fait avec la détection des états terminaux dans le graphe d'états. L'absence ou la présence de tels états ne garantit pas l'absence ou la présence d'exigence manquante. Donc, on peut utiliser les états terminaux comme indicateur, et non comme preuve, d'exigences manquantes, car si la spécification a des exigences manquantes et que le langage de modélisation oblige à rendre explicite toutes les transitions, certaines transitions n'existent pas et cela crée des états où aucune transition n'est activable (état terminal). À cause de cela, il faut faire attention durant la modélisation pour ne pas créer de transition qui a des préconditions trop faibles. Car elle risque d'être exécutée quand il ne devrait pas et cela pourrait masquer des états terminaux et ainsi ne permet plus de détecter les exigences manquantes de la spécification.

Le fait qu'il y ait des états terminaux peut indiquer plusieurs choses : qu'un bogue est présent dans le modèle, que la dernière fonctionnalité modélisée dépend d'une autre fonctionnalité qui n'est pas encore modélisée, que notre modèle est conçu pour s'arrêter, qu'il y ait une ambiguïté dans la spécification ou qu'un requit est manquant. C'est au concepteur de trouver la cause des états

terminaux. Si on utilise un langage de modélisation qui oblige de décrire tous les traitements de manière explicite comme discuté plus haut, on a plus de chance qu'une exigence manquante crée un état terminal, car il n'y a pas de comportement inféré implicitement.

La technique de détection des états terminaux la plus simple est de générer le graphe d'états et ensuite de le parcourir pour trouver les états qui n'ont pas d'état suivant. Ce sont les états terminaux. Cette technique fonctionne bien si le graphe d'états est petit. Par contre, si le graphe d'état est très grand, cela devient difficile, car il faut énumérer tous les états, avant de parcourir ce graphe pour trouver les états terminaux.

Il est possible d'utiliser le graphe de composantes fortement connexes pour trouver les états terminaux. Un état du graphe de composantes fortement connexes est un état terminal du graphe d'états si et seulement si le nœud du graphe de composantes fortement connexes est terminal et trivial [34].

Une catégorie de techniques intéressantes est celle à la volée (*on the fly*) [44]. Ce type de techniques vérifie une propriété pendant la génération du graphe d'états. Si la propriété est respectée, on génère tout le graphe d'états. Par contre, si la propriété n'est pas respectée, la vérification s'arrête à la découverte de l'échec de la propriété vérifiée, donc avant la génération complète du graphe d'états. Ceci permet de diminuer le temps de vérification en cas d'échec. Dans notre cas, on vérifie qu'il n'y a pas d'états terminaux. L'inconvénient est qu'on doit modifier le modèle de notre système de manière à ce que les états terminaux *désirables* ne soient pas confondus avec les états terminaux *non désirables*. Une autre manière est de complexifier la propriété pour vérifier si les états terminaux sont désirables ou non. Dans tous les cas, il est plus simple d'éviter de modéliser des états terminaux si ce n'est pas nécessaire. Le logiciel PROD fait cette vérification à la volée [45].

Il est possible de ne pas arrêter la vérification même si le modèle contient un état d'arrêt. Il suffit d'ajouter une boucle sans fin après l'état d'arrêt. Ainsi, cet état d'arrêt n'est pas un état terminal. Cela permet d'utiliser les techniques de vérification à la volée. Par contre, il faut faire attention à ne pas cacher d'états terminaux qui pourraient avoir une signification pour la vérification. Il est donc préférable de modéliser le système pour qu'il ne s'arrête jamais, plutôt que d'introduire ce genre de boucle.

3.5. Vérification de la cohérence

L'étape suivante est la vérification de la cohérence. On peut la faire avant ou après la vérification de la complétude. Pour vérifier la cohérence, on vérifie plusieurs propriétés qui sont nécessaires lors de la conception d'un modèle valide. Les propriétés qui sont indépendantes du modèle de calcul qu'on vérifie sont celles d'OPI (Obligation, Permission, Interdiction), d'équité et les invariants.

Parmi ces propriétés, il y a celles d'OPI [46]. Ces concepts viennent de la logique déontique, c'est-à-dire qui représente les alternatives d'une loi. Cette logique a été utilisée pour inférer des verdicts automatiquement [47] et est utilisé dans les systèmes multi agents [48]. Ces propriétés sont intéressantes, car si le modèle est modifié, il est possible de vérifier formellement qu'il n'y ait pas de comportement souhaité qui a été retiré par des modifications du modèle. Les propriétés d'obligations obligent qu'un certain comportement ou état arrive. Les propriétés de permission indiquent qu'il est possible de faire une chose, mais il n'est pas garanti qu'elles arrivent. On peut les utiliser pour vérifier que nos modifications n'enlèvent pas des options possibles qu'on souhaite vérifier. Pour les propriétés d'interdiction, c'est le contraire des propriétés d'obligation : on vérifie que cela n'arrive pas. Donc, s'il y a des comportements indésirables, on peut vérifier qu'ils n'arrivent pas même après des modifications du modèle. Les propriétés OPI peuvent être vérifiées avec les propriétés LTL/CTL, car c'est un sous-ensemble. On les inclut, car elles sont simples à ajouter pour un concepteur qui ne connaît pas les propriétés LTL ou CTL.

Plusieurs solutions existent pour vérifier la présence de contradictions dans un modèle. La première consiste à laisser l'outil qui génère le graphe d'états vérifier si les états générés sont valides. Par exemple, il peut vérifier que les valeurs mises dans une variable sont valides par rapport au type de la variable. Une autre manière de vérifier l'absence de contradiction est de vérifier la présence d'état terminal, car une contradiction empêche un changement d'état en rendant impossible la satisfaction de toutes les préconditions pour changer d'état. Ceci est détecté pendant la vérification de complétude.

Les propriétés d'équité[9] sont des propriétés sur les exécutions infinies du modèle dans le graphe d'états. Donc, elles ne donnent pas d'information sur les chemins finis. On dit qu'un arc du graphe d'états a une propriété d'équité si cette propriété est vraie pour l'arc dans tous les cycles infinis du modèle. Ces propriétés peuvent aussi être appliquées sur un ensemble d'arcs du graphe d'états. Un processus correspond à un tel ensemble dans le graphe d'états. Voici les trois propriétés d'équité en

ordre croissant de force : équité faible (just), équité forte (fair) et équité inconditionnel. Les propriétés sont inclusives. Si l'exécution d'un processus est équitable inconditionnellement, celle-ci est aussi équitable forte et équitable faible en même temps. De plus, si l'exécution est équitable forte, elle est aussi équitable faible.

Ces propriétés sont utiles pour vérifier l'absence de processus affamés. Un processus affamé est un processus qui est exécutable, et a un moment donné se voit refusé le droit d'être exécuter pour toujours. Ceci n'est pas un comportement souhaitable dans la majorité des systèmes, car le processus existe pour une raison et le fait qu'il ne soit plus jamais exécuté malgré qu'il soit exécutable amène une perte au processus global.

Un processus est équitable inconditionnellement si dans tous les cycles infinis du graphe d'états, il arrive une infinité de fois. Autrement dit, un tel processus est présent dans toutes les boucles infinies du modèle, donc si notre modèle boucle infiniment, les processus obligatoires dans la boucle sont inconditionnels. Par exemple, dans un modèle avec deux processus concurrents qui ne terminent jamais, toutes les exécutions possibles des deux processus doivent être équitables inconditionnellement, sinon un des processus peut être affamé.

Si l'on a deux processus et que l'un d'eux termine à un certain moment, la propriété d'équité inconditionnelle est toujours fausse pour le processus qui termine. Dans ce cas, si l'on veut vérifier que ce processus n'est pas affamé, on utilise la propriété d'équité faible. Pour qu'un processus soit équitable faiblement, il doit être exécuté dans tous les cycles infinis du graphe d'états où il est continuellement activable. Cela permet à un processus de devenir désactivé d'une manière permanente. On peut prendre comme exemple, un processus qui peut terminer définitivement. Cette propriété n'est pas assez forte pour garantir que des processus qui utilisent des sémaphores ne sont pas affamés. Ceci est causé par l'alternance d'activation des processus. Comme les processus sont activés en alternance, ils ne sont pas continuellement activables, donc la propriété ne s'applique pas.

On peut utiliser la propriété d'équité forte pour vérifier que des processus ne sont pas affamés s'ils utilisent des sémaphores. Un arc est équitable fortement s'il est infiniment activé lorsqu'il est infiniment activable dans les cycles infinis du graphe d'états. Donc, cela permet de vérifier que l'arc n'est pas affamé lorsque son activation est conditionnelle à un sémaphore, car cette propriété tient compte du comportement du sémaphore qui est d'activer et de désactiver en alternance l'arc.

On utilise ces propriétés d'équité de la manière suivante avec notre méthode de conception. L'ensemble des arcs d'un module doit être équitable inconditionnellement, car au moins un de ces arcs est exécuté à chaque cycle. Chaque arc à l'intérieur du module doit être équitable fortement ou faiblement. Si une de ces transitions n'est pas activable en même temps que d'autres, elle est équitable fortement, car quand elle est exécutable, elle est la seule qui peut être exécutée dans le module. Donc, si elle est exécutable une infinité de fois, elle doit être exécutée une infinité de fois. Si elle est concurrente (en conflit) avec d'autres transitions, elle est équitable faiblement. Plus d'information sur ces propriétés d'équité est disponible dans [49].

Tout modèle peut avoir des invariants. Un *invariant* est une propriété qui est constante dans tous les états du graphe d'état. Par exemple, on peut imaginer un système dans lequel deux signaux ont toujours la valeur inverse : un est à un et l'autre à zéro. Cette propriété constitue alors un invariant. On peut vérifier la cohérence du modèle en vérifiant les invariants. Certains algorithmes donnent des invariants existants d'un modèle pour certains modèles de calcul. On peut vérifier si les invariants du modèle y sont inclus et vérifier si les autres invariants ne sont pas le signe d'une incohérence.

Certains langages ou modèle de calcul de modélisation permettent de modéliser plus fidèlement un système et donc d'identifier plus finement les incohérences. Donc, il y a d'autres propriétés de cohérence qui existent, qui sont dépendantes du langage de modélisation ou du modèle de calcul. Ces propriétés et les implémentations des propriétés discutées ci-dessus sont présentées dans le chapitre 4.

3.6. Vérification de propriétés supplémentaires

Il est possible de vérifier des propriétés supplémentaires sur le modèle si elles sont écrites dans le langage LTL ou CTL. Pour ce faire, on crée des fonctions de navigation du graphe d'états ou on utilise un logiciel qui vérifie les formules exprimées dans ces logiques. On peut aussi faire la vérification à la volée des propriétés LTL. Une des difficultés pour la vérification des propriétés LTL provient de l'opérateur *suivant*. Cet opérateur oblige à garder tout le détail de l'espace d'état, ceci empêche l'utilisation de certaines optimisations du temps de calcul ou d'espace mémoire pour la vérification des propriétés.

Pour cette raison, certains logiciels ne supportent pas l'opérateur *suivant* des propriétés LTL. Il est possible de vérifier les propriétés LTL sans cet opérateur de manière plus rapide avec un graphe d'état

optimisé appelé *stubborn set* [50]. Cette technique est basée sur l'idée que si deux exécutions d'action sont assez similaires, il n'est pas nécessaire de vérifier les deux exécutions. Il est donc intéressant de créer une version sans cet opérateur des propriétés voulues pour faire une vérification rapide. Donc, si une de nos propriétés LTL est dans une liste connue [51, 52] de schémas de propriété et que ce schéma est aussi parmi ceux des propriétés CTL (ou une semblable) il est possible d'utiliser la version CTL pour accélérer la vérification. Par contre, cela offre une vérification plus faible pour certaines propriétés, donc il est possible de faire une première vérification avec les propriétés rapides et si elles sont toutes valides et ensuite de vérifier les propriétés complètes qui sont moins rapides.

Si on utilise un logiciel qui vérifie les propriétés LTL, il est intéressant de vérifier que l'automate de la propriété n'est pas trivialement vrai ou faux. Si l'automate est trivialement vrai ou faux, c'est une indication qu'il peut y avoir une erreur dans notre propriété.

3.7. Modification des propriétés LTL

Il y a deux raisons qui poussent à modifier les propriétés LTL : les états intermédiaires et pour accélérer la vitesse de vérification. Les états intermédiaires créent des problèmes seulement pour les propriétés synchronisées à une horloge. Pour régler ce problème, on pourrait créer un sous-graphe du graphe d'états qui contient seulement les états au front de l'horloge, mais ce n'est pas tous les outils qui implémentent cette fonctionnalité. Sinon, on doit modifier les propriétés LTL pour pouvoir les utiliser sur le graphe d'états à cause des états intermédiaires de calcul entre les états au front d'horloge. Comme plusieurs propriétés qu'on vérifie ne sont qu'au front d'horloge, on doit les modifier pour ignorer les états intermédiaires. En plus, en effectuant cette transformation, on peut obtenir une nouvelle propriété qui permet d'utiliser des algorithmes plus rapides de vérification si la nouvelle propriété n'a pas d'opérateur *suivant*. Un tel algorithme est les *stubborn set*.[50]

Une manière simple de faire cette modification de propriété LTL est de rajouter un opérateur logique ET aux propriétés atomiques avec une nouvelle propriété atomique *c*. Cette nouvelle propriété atomique est vraie seulement quand on est immédiatement après le coup de l'horloge. Par exemple, la propriété *c* peut être que la dernière transition exécutée est celle de l'horloge. Ensuite, il faut changer les opérateurs *suivant* pour des opérateurs *fatalement*. Il n'y a pas d'autre modification nécessaire à la propriété originale pour ignorer les états intermédiaires.

Prenons un exemple de transformation possible si on suit l'implémentation proposée dans ce chapitre. On suppose deux propriétés atomiques quelconques *p1* et *p2* et *c* la nouvelle propriété atomique décrite ci-haut. Alors, la propriété LTL de réponse globale $G(p1 \rightarrow Np2)$ peut être transformé en $G((p1 \wedge c) \rightarrow F(p2 \wedge c))$. La propriété initiale veut dire que peu importe quand *p1* est vrai à un front de l'horloge, alors *p2* est vrai au prochain front de l'horloge. La nouvelle propriété dit qu'éventuellement *p2* sera vrai après *p1* au front de l'horloge.

Par contre, cette transformation n'est pas optimale, car la nouvelle propriété est plus faible que l'originale, car on peut obtenir que la nouvelle propriété soit valide même si la propriété originale ne l'est pas. Un des avantages de cette nouvelle propriété est qu'elle enlève les opérateurs *suivants* et ainsi permet d'utiliser de meilleurs algorithmes pour toutes les propriétés. Par contre, il est suggéré de vérifier les propriétés transformées pour avoir des résultats partiels rapidement. Si ces propriétés ne trouvent aucune erreur, il faut vérifier avec les propriétés initiales pour une vérification complète.

On peut modifier autrement la propriété originale pour obtenir une propriété équivalente, mais il y a des opérateurs *suivants*. On ajoute un opérateur ET à chaque propriété atomique avec la propriété atomique *c*. On modifie les opérateurs *suivants*. Si la sous propriété suivante est de la forme Np, elle devient $N(\neg c \bigcup (p \wedge c))$. Si on reprend notre exemple avec la propriété initiale : $G(p1 \rightarrow Np2)$. On obtient la nouvelle propriété $G((p1 \wedge c) \rightarrow N(\neg c \bigcup (p2 \wedge c)))$. La propriété atomique *c* est vraie juste immédiatement après l'horloge, donc elle n'est pas vraie sur deux états consécutifs. Donc, la transformation force qu'on se déplace au front de l'horloge. Ceci est équivalent à la propriété originale. Par contre, elle conserve les opérateurs suivants de la propriété initiale. Donc, on peut utiliser les algorithmes optimisés seulement si la propriété initiale le peut.

3.8. Optimisation/abstraction du modèle

On a vu précédemment que l'espace d'états pouvait devenir très grand, ce qui engendre des difficultés pour effectuer des vérifications. Une technique d'optimisation manuelle de l'espace d'état est l'abstraction du modèle. Certains algorithmes de génération du graphe d'états peuvent compenser certaines abstractions automatiquement en créant des graphes d'états condensés, mais ils ne peuvent pas le faire aussi bien que si c'est fait manuellement. Par contre, si c'est fait manuellement il faut faire attention de ne pas perdre l'information nécessaire pour vérifier les propriétés voulues.

Une abstraction possible est de garder le minimum d'informations nécessaires dans le système. Une façon est d'avoir le minimum de conteneurs d'information (variables, places), quitte à en recalculer une partie avec l'information disponible, car plus on a de variables, plus il y a d'états possibles et plus il y a d'états qui sont distincts seulement par leur l'information supplémentaire. C'est au concepteur de trouver quelle information retirer. Cependant, le fait de modéliser de manière incrémentale permet au concepteur de voir l'effet de ces modifications sur l'espace d'états et peut le guider en lui indiquant les modifications qui augmentent beaucoup l'espace d'état.

Une fois qu'on a le minimum de conteneurs, on peut souvent abstraire davantage le modèle en faisant abstraction des valeurs des conteneurs. Cette abstraction est une diminution du nombre de valeurs possibles des types des variables. Cette diminution permet de diminuer le nombre d'états. Par exemple, un compteur d'attente peut être transformé en booléen qui indique seulement si on attend ou non au lieu du temps d'attente. La difficulté est de trouver les variables qui peuvent être abstraites sans perdre la validité de notre vérification.

Une autre approche pour diminuer la taille du graphe d'états est d'éviter de créer des états intermédiaires. Il y a souvent beaucoup d'états intermédiaires entre les coups d'horloge, entre autres à cause des différents ordres d'exécution des opérations parallèles, si on n'y fait pas attention. Il faut trouver un équilibre entre la diminution du nombre d'états intermédiaires et la complexité du modèle. Car diminuer le nombre d'états intermédiaires peut augmente la complexité du modèle en doublant certains calculs qui peuvent être factorisés si on ajoute des états intermédiaires.

S'il y a plusieurs processus en parallèle, cela crée une explosion combinatoire du nombre d'états, car on crée des états pour chaque ordre d'exécution possible. Si les modules ont beaucoup d'états intermédiaires dans leur calcul, cela amplifie ce problème. Il est possible de rendre l'exécution des modules séquentielle pour diminuer l'espace d'état. Il faut faire attention, car si on rend séquentiel l'ordre d'exécution des modules, on peut perdre des erreurs qui impliquent un ordre particulier d'exécution. Le projet Maria [53] propose une automatisation de cette technique qui ne pose pas de problème de perte de qualité de vérification. Pour ce faire, il se base sur le fait que les transitions internes des modules sont indépendantes l'une de l'autre. Donc, leur ordre d'exécution ne change rien. Quand c'est le cas, il n'est pas nécessaire de générer tous les ordres d'exécution possible.

Quand dans un état on peut choisir n'importe quelle valeur pour une variable, cela crée beaucoup

d'états suivants. En plus, tous ces états suivants en créent autant. C'est une source d'explosion combinatoire. Si la valeur n'a pas d'importance, on peut limiter cette explosion combinatoire en utilisant une valeur *don't care*. Ceci peut être une valeur supplémentaire définie dans le type de la variable. On peut aussi choisir une de ces valeurs qu'on utilise toujours quand la valeur n'a pas d'importance. Avec une de ces deux techniques, on évite la génération de toutes les valeurs permises quand cela n'est pas utile.

Le chapitre suivant décrit comme utiliser la méthodologie avec les réseaux de Petri colorés.

4. Méthodologie de modélisation

Ce chapitre décrit la façon d'utiliser la méthodologie présentée dans le chapitre 3 avec les réseaux de Petri colorés. On termine avec une description des vérifications formelles possibles avec les réseaux de Petri colorés.

4.1. Méthodologie

Les réseaux de Petri colorés permettent en théorie de faire toute la méthodologie présentée dans le chapitre 3. Par contre, les outils existants ne font pas toutes les vérifications et leur interopérabilité est plutôt limitée. Entre autres, on constatera que l'outil CPN Tools utilisé n'offre malheureusement pas beaucoup de support pour l'optimisation de l'analyse formelle.

Pour débuter, on considère le système à modéliser comme un ensemble de modules interconnectés. Pour modéliser le système, on crée d'abord des modules abstraits pour lesquels on définit une interface. Les modules sont alors reliés selon les dépendances qui existent entre eux. Ensuite, on modélise les fonctionnalités des modules de manière incrémentale. Dans le meilleur des cas, le comportement du module a déjà été décrit par une machine à états finis dans la spécification. La section suivante décrit la façon de modéliser un module à partir de cette machine à états finis. Sinon, il revient au concepteur d'élaborer lui-même la spécification du comportement du module directement en réseaux de Petri.

Dans la méthodologie, on tient compte du problème d'explosion combinatoire pendant la modélisation du système, en utilisant des techniques d'abstraction. Par conséquent, après chaque étape de la modélisation incrémentale, on regarde si l'augmentation du nombre d'états n'est pas trop grande ni disproportionnée. Si c'est le cas, on cherche à abstraire ce qui vient d'être modélisé pour diminuer le nombre d'états.

4.2. Méthode et algorithme de modélisation

La première étape pour modéliser la spécification en réseaux de Petri colorés est de créer une page par module et de créer l'interface des modules dans cette page. Ensuite, on crée l'interconnexion entre

les modules. Cela crée la structure des modules et permet la modélisation incrémentale des fonctionnalités de la spécification.

4.2.1. Structures des pages et des places des modules

La méthodologie de modélisation est basée sur une interconnexion de modules. Souvent, on trouve ces modules clairement identifiés dans la spécification. Si ce n'est pas le cas, c'est au concepteur de les identifier. Un module doit être instancié pour être utilisé dans le système et il peut l'être plusieurs fois. Les modules peuvent être séparés en deux catégories : synchrones et asynchrones. La technique de modélisation présentée ici supporte ces deux types de modules, dont la modélisation synchrone avec une horloge.

Les modules sont modélisés par les pages dans les réseaux de Petri colorés hiérarchiques. Chaque module peut avoir des places d'entrées, de sorties, de paramètres constants et d'états internes. Les places d'entrée et de sortie correspondent aux signaux d'entrée et de sortie définis dans l'interface du module. Pour permettre une meilleure réutilisation des modules, ceux-ci peuvent être paramétrés. Ces paramètres sont introduits au moyen de places qu'on appelle ici des places de paramètre, leur contenu est constant. Les transitions dans les modules vont consommer les jetons dans leurs places d'entrée quand ils arrivent et générer les jetons dans leurs places de sortie. L'arrivée des jetons est contrôlée à l'extérieur des modules.

La modélisation commence par l'identification des différents signaux à déclarer dans les interfaces des modules. On associe une couleur (un type) par signal en prenant soin de choisir une couleur dont le domaine de valeur est le plus petit possible (pour limiter l'explosion combinatoire). Par exemple, si un signal est représenté par un entier, on lui associe une couleur dénotant un intervalle d'entiers le plus petit possible. Ensuite, on crée une seule variable par couleur. Cela sert à donner le comportement voulu au modèle en utilisant les variables sur les étiquettes des arcs et dans les gardes des transitions. On crée des variables supplémentaires au besoin.

Par la suite on crée l'interface des modules. Dans leurs pages, on met une place par signal d'entrée de la couleur créée. On crée un port de type *in* pour ces places. On fait de même pour les signaux de sortie avec un port de type *out* sur les places. On crée aussi une place par paramètre dans les modules. Le type de ces places est de type *in/out,* car quand on lit la valeur dans la place, en fait on enlève le

jeton et on en remet un nouveau avec la même valeur.

Ensuite, on crée la page supérieure qui contient le système et relie les modules entre eux pour leur permettre de communiquer entre eux. Dans cette page, on crée une transition de substitution par module asynchrone qu'on lie hiérarchiquement à cette page. On crée une place avec le type correspondant par signal d'entrée de nos transitions qu'on vient de créer. On crée une place par module même si un signal est diffusé à plusieurs destinations. On crée un arc de chaque place vers le module correspondant. On relie la place de la page supérieure avec la place de la sous-page. On fait de même pour les signaux de sortie.

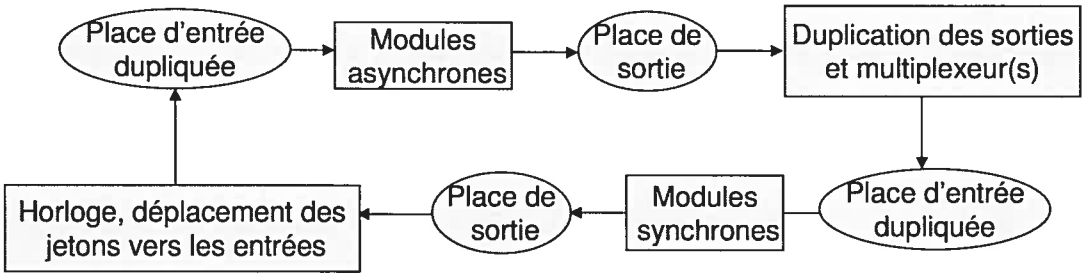

Figure 13 : Plan du modèle

On crée une place par signal d'entrée des modules, même si la valeur est diffusée, car dans le fonctionnement du modèle, chaque module essaie de prendre les jetons dans leurs places d'entrées. Et par conséquent, si cette place était partagée entre plusieurs modules, cela permettrait à un seul des modules de prendre les jetons de cette place. Il est possible de d'optimiser cela comme décrit dans la section 4.4.1.

Ensuite, on ajoute les modules synchrones dans la page supérieure. On utilise comme place d'entrée des modules les places de sortie des modules asynchrones si ce signal n'est pas diffusé. Si le signal est diffusé, on crée une place pour chaque destination du signal. Ensuite, on crée une transition pour dupliquer la place du module asynchrone vers toutes les places créées.

Toujours dans la page supérieure, on crée une transition qui représente l'horloge. Cette transition, lorsqu'elle sera déclenchée, transférera les jetons (signaux) de sortie vers les jetons (signaux) d'entrée de tous les modules. Elle fera les duplications nécessaires pour les diffusions. Elle correspond au front ascendant ou descendant de l'horloge.

Ensuite, on recommence pour les autres niveaux de hiérarchie, s'il y a des modules imbriqués. Il existe deux types de modules : ceux qui ont juste une instance et ceux qui en ont plusieurs. Dans le cas des modules qui ont une seule instance, celle-ci correspond à la page du module. Quand on mentionnera la page d'une instance de module, ce sera la page du module s'il n'y en a qu'une.

Dans le cas des modules comportant plusieurs instances, on propose deux manières de les implémenter. La première est de créer une page qui décrit ce module et de l'instancier plusieurs fois. La deuxième est de créer une page où toutes les instances du module sont décrites. L'avantage de la première est sa simplicité, car elle ressemble à ce qu'on a l'habitude de faire. L'avantage de la deuxième est que l'on voit l'état de toutes les instances dans la même page et qu'il est facile d'utiliser des variables pour configurer le nombre d'instances. Il serait possible de créer ou détruire des instances dynamiquement avec la deuxième manière, ce qui est impossible avec la première. Cette création/destruction dynamique n'est pas nécessaire ni possible en pratique. Par contre, cela est possible avec de nouvelles technologies comme les FPGA reconfigurables dynamiquement. Malheureusement, la deuxième approche ne peut pas être utilisée si les instances sont réparties un peu partout dans le modèle. C'est utile seulement si les instances sont toutes au même endroit, comme lorsqu'on a un tableau d'instances.

Les places d'entrée et de sortie des pages de module doivent être modifiées s'il y a plus qu'une instance du module ou s'il y a création dynamique de module. On utilise comme couleur des places une structure à deux éléments. Le premier élément est le numéro de l'instance de la donnée et le deuxième correspond à la donnée elle-même.

4.2.2. Comportement des modules

On suppose que chaque module possède une description de son comportement sous forme de machine à états finis. Soit la spécification donne cette description, soit le concepteur est capable de la créer. La machine à états peut avoir un état interne qui détermine son comportement. Cet état interne est gardé dans des places dans la page du module. On essaye de garder un seul jeton à la fois dans les places pour que le modèle soit *1 safe*. Si le concepteur implémente plusieurs instances du module dans la même page, il doit modifier en conséquence le modèle du module. Il est conseillé d'utiliser une hiérarchie interne si la machine à états prend beaucoup d'espace sur une page. Ceci facilite la conception, car on voit mieux ce qui se passe.

L'implémentation la plus directe du FSM d'un module est de mettre une transition dans le module par transition dans la machine à états finis. Ceci crée un seul niveau de transition. Sur les arcs entrants des transitions, on trouve les préconditions et sur les arcs sortants les postconditions de la machine à état. Par contre, cela n'est pas nécessairement la meilleure implémentation, car elle peut créer beaucoup plus de transitions que nécessaire et cela complique la modification du modèle durant l'ajout de fonctionnalités. Il est parfois utile de créer des états intermédiaires durant le calcul des jetons de sortie en factorisant le calcul redondant dans plusieurs transitions. L'avantage principal de l'implémentation directe est justement qu'elle ne crée pas d'état intermédiaire, et ainsi diminue la taille du graphe d'état. C'est au concepteur de trouver le bon équilibre entre le nombre d'états et la complexité de modifier le modèle.

4.2.3. Autres remarques concernant la modélisation

Il y a trois remarques concernant la modélisation qui peuvent aider le concepteur. La première porte sur l'ajout rapide de fonctionnalités dans le modèle. Pour ce faire, on peut retirer du modèle certaines options redondantes ou similaires pendant la conception. Ceci diminue la taille du graphe d'états et accélère la vérification à chaque incrément. Par contre, il ne faut pas oublier de remettre toutes les options pour la vérification finale. Il est aussi conseillé de vérifier toutes les options de temps en temps pour découvrir plus rapidement les problèmes s'il y en a parmi les options qui sont retirés. Cette diminution peut être faite avec des variables globales ou avec plusieurs versions des couleurs ayant un nombre de valeurs possibles différent. Aussi, avoir un seul endroit où on peut modifier ces paramètres diminue la chance d'oublier certaines diminutions qui ont été faites.

La deuxième remarque est qu'il est conseillé de donner un nom révélateur aux groupes page/transition. Cela permet de suivre facilement le fonctionnement du système en regardant la séquence de transitions activée. Si on ne le fait pas, il faut connaître les valeurs des variables prises pendant l'activation de la transition pour savoir ce qui est fait dans le modèle. Utiliser de tels groupes peut créer plus de transitions parallèles que le minimum requis, mais n'augmente pas le nombre d'états, car les transitions ajoutées sont en conflit avec celles qui existantes. C'est encore au concepteur de trouver l'équilibre entre le nombre de transitions et la facilité de suivre le déroulement d'un système avec une séquence de transitions effectuées.

La dernière remarque est qu'après une modification du modèle et avant de lancer la génération du

graphe d'états, on peut faire une simulation rapide. Cela peut servir à vérifier si on ne tombe pas rapidement dans une inconsistance du modèle ou un état terminal. Faire une simulation rapide est plus rapide que de générer le graphe d'état.

4.3. Vérification formelle en réseau de Petri

Il y a trois méthodes pour faire de la vérification formelle d'un réseau de Petri. On peut analyser soit le graphe d'états, le graphe de composantes fortement connexes ou la structure du modèle.

Pour l'analyse du graphe d'états, on passe par la génération d'un graphe qui donne les états atteignables à partir de l'état initial du modèle. Dans notre méthodologie, l'analyse de ce graphe est utilisée pour vérifier des propriétés dont certaines concernent la vérification de la complétude, la vérification de la cohérence et la vérification des propriétés spécifiées par le concepteur.

L'utilisation du graphe de composantes fortement connexes est utilisée pour réduire le temps de vérification de certaines propriétés. Il y a deux raisons 1) le graphe est plus petit et 2) pour certaines propriétés, les algorithmes ont une complexité inférieur.[54]

L'analyse structurelle est une analyse ne prenant en compte que de la structure du modèle, (c.-à-d. indépendante de l'état initial) pour trouver certaines propriétés du modèle qui sont indépendantes de l'état. Lorsque pertinente, cette approche permet d'éviter le problème d'explosion combinatoire du nombre d'états.

On suppose que le modèle est syntaxiquement correct, c'est-à-dire qu'il respecte la définition d'un réseau de Petri coloré. L'outil utilisé fait cette vérification.

4.3.1. Analyse possible

Il est possible de vérifier plusieurs types de propriétés avec les réseaux de Petri. Avec le graphe d'états, on peut détecter des états terminaux comme décris dans le chapitre 3. On peut vérifier les propriétés LTL et CTL. On peut aussi vérifier si les transitions sont mortes ou vivantes. On peut vérifier la quantité maximale et minimale du nombre de jetons dans une place de manière exacte. On peut trouver le meilleur ensemble multiple dans une place. On peut aussi trouver la liste des transitions concurrente et celle en conflit.

Si on utilise un graphe des composantes fortement connexes, on peut vérifier plus rapidement certaines propriétés, car le graphe est souvent plus petit que le graphe d'états. Dans notre méthodologie, les vérifications qui peuvent être accélérées grâce à ce graphe sont la détection des états terminaux, la vérification d'un sous-ensemble de propriétés CTL, la détection des transitions vivantes et la vérification des propriétés d'équité.

On peut avoir des statistiques de ces deux graphes, comme le nombre d'arcs et le nombre de nœuds pour chacun. Ceci peut être utilisé pour vérifier la cohérence du modèle. Il y a généralement beaucoup moins de nœuds dans le graphe de composantes fortement connexes que dans le graphe d'états, car les états faisant parti d'un cycle sont regroupés dans une même composante fortement connexe et donc dans un même nœud du graphe.

Avec l'analyse structurelle, on peut vérifier si des transitions n'ont aucune précondition ou aucune postcondition. On peut vérifier qu'il n'y a pas de sous-hiérarchie (page) vide. On peut calculer le nombre maximum de jetons pouvant se trouver dans une place du réseau. C'est un maximum structurel qui est une borne supérieure de ce qui est trouvé avec le graphe d'état. On peut aussi calculer des invariants de place et de transitions.

4.3.2. Vérification de la finitude

Avec les réseaux de Petri, on peut vérifier la finitude du modèle dans certains cas même si le problème général est indécidable. La manière la plus simple est de générer le graphe d'états. Si la génération arrête, c'est que le graphe est fini. Sinon, la cause peut être que le graphe est gros et va finir dans le futur ou que le graphe est infini. Cela rend cette manière de vérifier la finitude du graphe d'états utilisable que pour des graphes qui sont petits.

Une deuxième manière de détecter la finitude du graphe d'états est basée sur l'analyse des endroits qui peuvent créer une infinité d'états. Dans les réseaux de Petri colorés, on a des couleurs avec une quantité finie de valeurs possibles. On a aussi un nombre fini de places. Ce qui peut créer un nombre infini d'états atteignables est une place avec un nombre de jetons qui augmente toujours sans limites. Une manière simple d'avoir une telle place, est de rajouter un jeton sans en enlever dans une place. Une version plus compliquée est d'utiliser des listes de jetons dans les places, car une place avec une liste d'éléments peut avoir un nombre variable d'éléments dans la liste et un seul jeton. Donc, si on en

ajoute toujours des jetons, on a un système avec une infinité d'états. Heureusement, un modèle avec une borne maximale finie du nombre de jetons, idem pour les listes, garantit d'avoir un nombre fini d'états.

On peut vérifier la borne du nombre de jetons dans les places par le graphe d'état. Mais si on veut vérifier que le modèle est fini avant de générer le graphe d'états on doit trouver d'autres moyens. Pour ce faire, il reste l'analyse de la structure du modèle. Une des propriétés que cette analyse peut donner est une borne structurelle du nombre de jetons dans les places, maximal et minimal. Cette borne structurelle correspond à une borne supérieure ou inférieure de jetons dans le graphe d'états. Elle peut ne pas être atteinte dans le graphe d'état. Par contre, si cette borne est finie, on a la preuve que l'espace d'états est fini. Aussi, si on utilise la méthodologie de modélisation donnée dans ce mémoire, la borne maximale est de 1 pour toutes les places.

Par contre, cette borne ne tient pas compte du nombre de jetons dans les listes. On pourrait modifier cet algorithme pour qu'il regarde le nombre d'éléments dans les listes au lieu du nombre d'éléments dans les places. Si le concepteur n'utilise pas de listes ou s'il s'assure que la taille de ces listes est finie, on peut utiliser la détection des bornes structurelle pour vérifier qu'un modèle est fini.

4.3.3. Vérification de la complétude

La complétude pour les réseaux de Petri peut être vérifiée avec la détection d'états terminaux dans le graphe d'états. Cela peut être accéléré avec le graphe de composantes fortement connexes, comme expliqué dans la section 3.4.

4.3.4. Vérification de la cohérence

On vérifie la cohérence du système en vérifiant des propriétés sur son modèle en réseau de Petri. Certaines propriétés peuvent être vérifiées par plusieurs techniques. Lorsque c'est le cas, on indique la technique la plus rapide à notre connaissance pour vérifier cette propriété. Avec le graphe d'états, on peut vérifier les transitions mortes, le nombre maximal et minimal de jetons des places, les transitions concurrentes et en conflits. Le graphe de composantes fortement connexes permet de trouver les transitions vivantes et de vérifier les propriétés d'équités. Finalement, l'analyse structurelle peut être utilisée pour trouver les invariants de place et de transitions.

Avec le graphe d'états, on peut vérifier s'il y a des transitions mortes : c'est-à-dire des transitions qui ne sont jamais activables. Ces transitions sont inutiles dans le modèle. Soit la transition est activable dans certaines conditions soit elle n'a aucune raison d'exister.

Par ailleurs, on peut calculer une borne sur le nombre maximal ou minimal de jetons pouvant se trouver dans une place du réseau. Si on suit la méthode proposée, il doit y avoir au maximum un jeton par place. Donc, on peut utiliser cette vérification pour détecter des problèmes avec le modèle. De plus les places de paramètres doivent avoir au minimum et maximum un jeton et sa valeur doit être constante. Pour les places d'entrée et de sorties des modules, le minimum de jetons doit être de 0. Pour les places internes des modules, il est conseillé d'implémenter au maximum 1 jeton pour avoir un réseau de Petri *1 safe*, car il y a plus d'outils sur cette catégorie de réseau de Petri et ils ont tendance à être plus performants.

Toujours avec le graphe d'états, on peut aussi lister les transitions concurrentes, activables en même temps, et vérifier si ce comportement est permis dans la spécification. On peut aussi lister les transitions mutuellement exclusives, que l'on dit structurellement conflictuelles. Deux transitions sont structurellement conflictuelles si elles partagent au moins une place d'entrée. Le cas échéant, le tir d'une des transitions risque de compromettre le tir de la seconde s'il n'y a pas suffisamment de jetons dans la place partagée. Des transitions en conflit sont des transitions qui sont activables en même temps. Par exemple, si une de ces transitions est activée, celles en conflit ne sont plus activables. La Figure 14 montre deux transitions T1 et T2 en conflit.

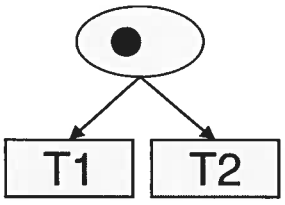

Figure 14 : Réseau de Petri Place/Transition avec conflit

Si dans la spécification on a deux transitions qui doivent être concurrentes, elles ne doivent pas être en conflit dans le modèle. Si elles sont en conflit, on ne vérifie pas tous les ordres d'exécution possible de ces transitions.

Pour vérifier la cohérence de la spécification, on peut vérifier les propriétés d'équité (décrites dans

la section 3.5) pour les transitions. L'équité inconditionnelle impose les contraintes les plus fortes sur les exécutions considérées. Les contraintes portent sur le franchissement équitable des transitions et peuvent donc être vérifiées en parcourant les arcs des graphes d'états ou de composantes fortement connexes. Dans notre cas, la vérification consiste à s'assurer que le modèle permet toujours le futur franchissement de chacune des transitions du système, c'est-à-dire le perpétuel déclenchement « tôt ou tard » de chacun des processus.

Ces propriétés d'équité sont définies pour l'activation d'une transition dans une trace d'exécution. On dit que la transition possède cette propriété si toutes les traces d'exécutions ont cette propriété pour cette transition.

La vérification des propriétés d'équité permet ainsi d'identifier les exécutions où certaines transitions (processus) ne sont jamais effectivement tirées alors qu'elles sont pourtant franchissables. Ainsi, dans notre méthode de conception, le franchissement d'une transition est inconditionnellement équitable si elle est toujours fatalement activée dans toutes les traces d'exécutions. Par exemple, la transition d'horloge est toujours exécutée une infinité de fois dans les cycles infinis. Par ailleurs, on dira que l'ensemble des transitions d'un module doit être inconditionnellement équitable, car il doit y en avoir au moins une qui est exécutée à chaque cycle. Chaque transition à l'intérieur du module doit être équitable faiblement ou fortement. Si une transition d'un module n'est pas censée être en concurrence à d'autres transitions dans le même module, elle doit être équitablement forte. Si elle est en conflit avec d'autres transitions, elle est équitablement faible.

Finalement, le graphe de composantes fortement connexes doit avoir un seul ou peu d'états si le modèle est cyclique, car il peut y avoir des états supplémentaires pour l'initialisation ou comme des états terminaux désirables.

On peut aussi déterminer les transitions qui sont vivantes. Une transition vivante est une transition qui est activable dans toutes les composantes fortement connexes. En d'autres mots, pour une transition vivante, il est toujours possible dans le futur qu'elle soit activée : on ne peut pas empêcher son tir en permanence. Il est normal qu'une transition soit vivante, car notre modèle est cyclique. Il est donc plus pertinent de vérifier les transitions non vivantes. Une transition non vivante n'est pas nécessairement une transition morte, car elle pourrait n'être activable que dans certaines traces d'exécution. Toutes les transitions dans nos modules devraient être vivantes si notre modèle est

cyclique sauf pour les transitions d'initialisation (s'il y en a) et les transitions qui peuvent être désactivées en permanence.

L'analyse de la structure du modèle permet d'identifier les transitions qui n'ont aucune précondition ou postcondition. Si une transition n'a aucune précondition, c'est qu'elle peut être toujours activée. Ce n'est pas souhaitable, car si la transition a des postconditions, cela crée un modèle infini. Pour les postconditions, il est possible qu'une transition n'en ait pas, mais notre approche de modélisation permet d'éviter ce cas. À noter qu'il est également possible de vérifier qu'aucune page n'est vide. La présence d'une page vide signifierait qu'on a oublié d'implémenter un module.

L'analyse structurelle d'un réseau de Petri permet également de calculer automatiquement un certain nombre d'invariants. Il y a deux types d'invariants, les invariants de places et ceux de transitions. On peut vérifier si la liste des invariants contient tous ceux attendus et on peut vérifier si les autres sont corrects. Entre autres, tous les modèles devraient présenter un invariant sur les places dénotant les paramètres, puisque ceux-ci ne changent pas pendant l'exécution [1].

4.3.5. Vérification de propriétés supplémentaires

Avec les réseaux de Petri, il est possible de vérifier des propriétés supplémentaires proposées par le concepteur. Bien que d'autres formalismes puissent être utilisés, on suggère de spécifier ces propriétés en logique temporelle CTL ou LTL dont l'expressivité est généralement suffisante. À noter que les auteurs de l'outil CPN Tools ont développé une telle extension appelée ASKCTL [54] qui permet de vérifier des formules CTL sur un réseau de Petri coloré en passant par son graphe d'états. Ils expliquent qu'un sous-ensemble de CTL peut également être vérifié à partir du graphe de composantes fortement connexes [55].

Pour vérifier les propriétés supplémentaires, on doit résoudre un problème : il y a des états intermédiaires entre les fronts de l'horloge dans le graphe d'état. Ceci est problématique, car plusieurs propriétés intéressantes pour le concepteur doivent être vérifiées uniquement au front de l'horloge. Pour ce faire, on doit parcourir le graphe pour ignorer les états intermédiaires. Ceci est expliqué dans la section 3.7.

4.4. Diminution du temps de vérification

Il y a plusieurs moyens pour réduire le temps de vérification du modèle. On peut utiliser des outils qui font des optimisations automatiquement. On peut réduire le nombre de places, réduire l'espace d'états ou optimiser la représentation du graphe d'états. C'est ce dont on discute dans les prochaines sous-sections. On peut aussi décomposer les propriétés pour pouvoir en vérifier certaines parties plus rapidement. Mentionnons qu'il est souvent difficile de réutiliser les outils d'optimisation développés pour d'autres *framework* de vérification. En effet, l'interopérabilité est limitée à cause des formats de représentation utilisés par ces outils, mais aussi à cause des différences entre les types de réseaux de Petri considérés par chacun. Le standard PNML [28], en cours de définition, a justement été proposé pour régler le problème de format de fichiers à partager entre les outils de réseaux de Petri.

Tel que discuté dans le chapitre 3, des stratégies de modélisation ad hoc, visant à augmenter l'abstraction du modèle, peuvent être appliquées pour faciliter la vérification des réseaux de Petri en un temps raisonnable.

4.4.1. Réduction du nombre de places

Le nombre de places ne change pas directement le nombre d'états, mais il a un impact sur le temps de calcul de l'espace d'états. Plus il y a de places pour un même nombre d'états, plus long est le temps de calcul. Ceci est causé par la comparaison de deux états qui est une opération fréquente et obligatoire dans les algorithmes de vérification. Il y a trois manières principales de représenter les états : par diagramme de décisions binaires (*Binary decision diagram*, BDD)) [10, 56], par objet et par table de hachage. L'impacte de la réduction du nombre de places dépend du type de représentation utilisé, mais plus il y a de places, plus c'est long. Les techniques sont expliquées plus en détail dans la section 4.4.3.

Le nombre de places peut avoir un impact sur le temps de calcul du graphe d'états, mais aussi sur l'espace mémoire utilisé pour l'enregistrer. Les algorithmes de vérification doivent déterminer si le prochain état calculé a déjà été traité. Donc, on doit pouvoir savoir si un état est dans l'ensemble des états déjà trouvés. En plus, le nombre de places a un impact sur la taille-mémoire nécessaire pour sauvegarder un état seulement si sa représentation est sous forme de BDD ou d'objet.

L'impact du nombre de places sur le temps de calcul du graphe d'état se fait ressentir sur les trois représentations à des niveaux différents. Si la représentation est sous forme de table de hachage, le nombre de places a un impact linéaire sur le temps de calcul, car le temps de calcul du code de hachage est linéaire sur le nombre de places. Avec une représentation sous forme d'objet on doit tenir compte que la comparaison de deux états est faite plusieurs fois et que cette comparaison est linéaire sur le nombre de place. Donc, pour savoir l'impact total du nombre de place p, on doit multiplier p par le nombre de comparaisons pour savoir si un état est dans l'ensemble des états déjà vérifiés. Donc, si la représentation est sous forme d'une liste, l'impact total du nombre de places est d'ordre p^2 Si la représentation est sous forme d'arbre auto balancé, l'impact est d'ordre $p \log p$.

L'outil CPN tool qu'on utilise représente les états sous forme d'objets et les gardes dans une structure arborescente auto balancée. Donc, l'impact du nombre de places envisagé pour notre étude de cas est d'ordre $p \log p$.

À noter qu'on peut diminuer le nombre de places avec la technique de modélisation décrite, quand un signal est diffusé, en éliminant les places supplémentaires. Ensuite, les modules doivent utiliser des arcs bidirectionnels sur cette place commune pour lire la valeur du signal au lieu de prendre le jeton. C'est la transition de l'horloge qui enlève le jeton de cette place commune. Ce changement diminue le nombre de places nécessaires sans augmenter le nombre d'états et diminue ainsi le temps de vérification. Par contre, il est moins systématique et on ne peut pas se fier au fait que cette place contient un jeton juste quand on doit exécuter la page. Cela peut modifier le module.

On peut aussi inclure plusieurs instances d'un module dans la même page. Cela diminue le nombre de places (puisque les places représentant des valeurs partagées par ces modules sont fusionnées en places uniques) et permet d'avoir un nombre paramétrable d'instances. En plus, on voit dans la même page l'état de toutes les instances de ces modules. Cela simplifie la modélisation d'un multiplexeur qui prend en entrée un signal de toutes les instances. On peut aussi diminuer la taille du graphe d'états en n'activant que l'instance qui est sélectionnée par le multiplexeur si cela ne change pas le comportement du modèle.

4.4.2. Réduction de l'espace d'états

Pour réduire l'espace d'états, on peut faire une abstraction du modèle telle que celle décrite dans la

section 3.8. Voici d'autres techniques qui peuvent être utilisées.

Une manière de diminuer l'information qui est conservée dans un module, donc de l'espace d'états, est de diminuer le nombre de places internes de ce module (si cela n'a pas encore été fait) en éliminant les places vides par exemple. On ne peut pas vérifier directement si une place est vide avec les réseaux de Petri colorés (il faudrait des arcs inhibiteurs). Si on a besoin de savoir qu'une place est vide, on peut utiliser une liste dans la place. Cette liste contient les jetons de la place et peut être vide. Il est possible de détecter les listes vides.

Dans l'étude de cas, on doit modéliser un multiplexeur. Il est possible de les modéliser de plusieurs manières et l'une d'elle permet de réduire l'espace d'états. La première manière est d'utiliser une place pour chaque entrée et pour la sortie ainsi qu'une seule transition. La transition prend les jetons dans les places d'entrer et met le bon jeton dans la place de sortie. Pour optimiser la représentation du multiplexeur, on peut éliminer la transition sans modifier le comportement du modèle. Pour ce faire, on enlève les places d'entrée du multiplexeur et on modifie les transitions qui mettent des jetons dans ces places. Avec une fonction SML, il est possible que ces transitions mettent directement un jeton dans la place destination du multiplexeur quand il est choisi par le multiplexeur. Pour utiliser cette technique de réduction de l'espace d'états, il faut savoir quel jeton d'entrée est choisi par le multiplexeur avant que les transitions ne mettent les jetons dans les places d'entrée du multiplexeur. La transition en moins diminue le nombre d'états intermédiaires.

Cette modification des transitions revient à diminuer le nombre de jetons qu'une transition dépose dans des places. On peut aller plus loin et ne pas activer les modules si leur exécution n'est pas utile, comme quand elles ne sont pas sélectionnées par le multiplexeur.

4.4.3. Optimisation par les outils

Certains outils mettent en œuvre différents types d'optimisations, dont l'optimisation de la représentation du graphe d'états, diminuer la taille du graphe d'états et la vérification à la volée. On peut en représenter les états sous forme de BDD, de table de hachage ou sous forme d'objets gardés dans une liste ou un arbre.

La représentation du graphe d'états sous forme de liste ou d'arbre d'objets nécessite plus de mémoire que la représentation utilisant une table de hachage ou un BDD. La complexité des

algorithmes de recherche et d'ajout d'états dans une liste ou un arbre d'objets est plus grande que celle des tables de hachage. Plus précisément, étant donné une représentation sous forme d'objets, la recherche permettant de déterminer si un état a déjà été visité est 1) d'ordre linéaire en fonction du nombre de places si on utilise des listes ou 2) d'ordre logarithmique si on utilise une arborescence. Comme cette recherche doit se faire pour chaque état, le nombre d'états n à un impacte d'ordre exponentiel pour les listes et $n \log n$ pour les arbres.

Étant donné une représentation utilisant une table de hachage, la recherche d'un état du graphe est d'ordre constant et impose donc un coût constant sur le temps de vérification. L'espace mémoire nécessaire pour stocker la table n'est toutefois pas plus petit que dans le cas des listes/arbres d'objets. Par contre, la table de hachage peut s'avère utile pour vérifier des propriétés à la volée : dans ce cas, le graphe est généré au fur et à mesure de la vérification et on ne stocke dans la table que les états visités. En plus de profiter d'une recherche en temps constant, on peut aussi limiter l'espace occupé par la table en ne stockant que la clef de hachage des états visités (et non tout le contenu de l'état). Par contre, cela empêche de détecter des collisions dans la table de hachage et ainsi il est possible que certains états ne soient pas vérifiés.

Quant aux BDD, ils permettent de représenter de manière compacte et efficace les états ou des ensembles d'états, les systèmes de transitions et les formules. Il s'agit de graphes dirigés acycliques ayant un seul nœud racine. Les nœuds terminaux sont étiquetés par 0 ou 1 alors que les nœuds non terminaux sont étiquetés par les variables du modèle. Chaque niveau de l'arbre est associé à une variable booléenne donnée et à la décision de la valeur qu'on doit lui attribuer. Ainsi, chaque nœud non terminal (p. ex. variable x) a deux arcs sortants (p. ex. un pour le cas où la variable x vaut 0, l'autre pour le cas où elle vaut 1) qui vont au prochain niveau (p. ex. décision pour une autre variable y). Ainsi, pour trouver un état, il suffit de consulter un chemin le représentant dans le BDD. Sachant qu'un état est décrit par les valeurs associées aux variables du modèle, il suffit, à chaque niveau du BDD, de suivre l'arc correspondant à la valeur de la variable considérée. Lorsqu'on atteint une feuille, si l'étiquette vaut 1, alors on conclut que l'état est trouvé, sinon l'état est absent.

Si on ne fait pas d'optimisation sur les BDD, l'espace mémoire nécessaire est le même que celui des tables de vérité. Par contre, il est facile d'éliminer les redondances dans cette représentation et ainsi de diminuer la taille mémoire requise. Une de ces optimisations consiste à réutiliser les sous-arbres qui sont identiques au lieu de les dupliquer. Beaucoup de travaux de recherche actuels semblent

privilégier l'emploi des BDD.

En ce qui concerne l'outil CPN Tools, il utilise une représentation des états sous forme d'objets organisés en arborescence. Ceci n'est certes pas la technique la plus économique en coût et en temps, mais elle a l'avantage d'être simple. De plus, comme on souhaite conserver la valeur de l'état de l'objet (et non seulement la clef l'identifiant), la représentation par table de hachage ne serait pas avantageuse du coté de la quantité de mémoire nécessaire.

Certains outils permettent de réduire la taille des graphes d'états en remplaçant, par un état unique abstrait, un ensemble d'états partageant une propriété commune, c'est-à-dire formant une classe d'équivalence. Cette réduction par abstraction peut bien sûr entraîner une perte d'information. Le graphe de composantes fortement connexes est un exemple de réduction du graphe d'états où il y a possiblement perte d'information. Mais si on s'intéresse à évaluer les propriétés d'atteignabilité, l'information contenue dans ce type de graphe est généralement suffisante.

Finalement, la vérification à la volée décrite dans la section 3.4 peut être utilisée en réseau de Petri. L'outil CPN-AMI l'utilise [14].

5. Étude de cas

Pour valider la méthodologie présentée, on modélise la spécification du bus AHB-Lite [57] avec les réseaux de Petri colorés dans l'outil CPN Tools. Ce bus est une version simplifiée du bus AHB [58] (Advanced High-performance Bus). Le modèle a été construit de manière incrémentale. Suite à chaque incrément, nous avons procédé à la vérification de la finitude, de la complétude et de la cohérence du modèle, tout en résolvant (au moyen d'abstractions) les problèmes d'explosion combinatoire. Plusieurs autres propriétés spécifiquement définies pour le bus AHB-Lite ont ensuite été vérifiées sur le modèle final.

Ces propriétés spécifiques ont été formulées par une compagnie bien connue dans le domaine de l'EDA (conception électronique assistée par ordinateur). On nous a ainsi soumis 55 propriétés couvrant la spécification du bus AHB. Puisque notre étude de cas ne que concerne que la spécification du bus AHB-Lite, seulement un sous-ensemble de ces propriétés a été utilisé. Néanmoins, notre étude considère un plus grand nombre de propriétés que la plupart des autres travaux publiés sur la vérification formelle du bus AHB.

La méthodologie que nous avons proposée est principalement destinée à la vérification de systèmes embarqués. Elle considère donc à la fois une partie matérielle et une partie logicielle. Bien qu'elle ne s'intéresse pas à la partie logicielle en tant que telle, notre étude de cas demeure particulièrement intéressante puisque peu d'expériences de modélisation de systèmes matériels ont été réalisées au moyen des réseaux de Petri colorés [59]. En revanche, les exemples de modélisation de systèmes logiciels utilisant ce formalisme sont beaucoup plus nombreux [60].

5.1. AMBA, AHB et AHB-Lite

La spécification *Advanced Microcontroller Bus Architecture* (AMBA) est une spécification de bus matériel faite par la compagnie ARM. En fait, cette spécification contient la définition de trois versions de bus : *Advanced Peripheral Bus* (APB), *Advanced System Bus* (ASB) et *Advanced High-performance Bus* (AHB). Le bus APB est un bus de communication avec des périphériques lents. Il est optimisé pour consommer peu d'énergie, mais il transfère l'information moins vite que les deux autres bus. Pour implémenter le bus principal d'un système, un bus ASB ou AHB est recommandé. Le

bus AHB est un bus plus compliqué, mais plus performant que les autres. On peut faire un pont vers le bus APB depuis le bus ASB ou AHB.

Le bus AHB présente une architecture pipeline en deux étages traitant respectivement l'*adresse* et la *donnée* du transfert. En plus, il offre plusieurs types de transfert dont un en rafale.

Tel que mentionné au début de ce chapitre, la spécification du bus AHB-Lite est une version simplifiée de celle du bus AHB (Figure 15) et c'est celle-ci qui a été considérée pour l'étude de cas. Cette version simplifiée a l'avantage de simplifier la conception de l'interface des modules avec ce bus. Du reste, on peut ajouter une enveloppe (wrapper) standard pour assurer la compatibilité de cette interface simplifiée avec la spécification complète si cela est éventuellement nécessaire. À noter que la version non simplifiée du bus présente au moins un cas d'interblocage, avec les transferts de type SPLIT, qui n'est pas présent dans la version simplifiée. Celui-ci ne fera donc pas l'objet de notre vérification.

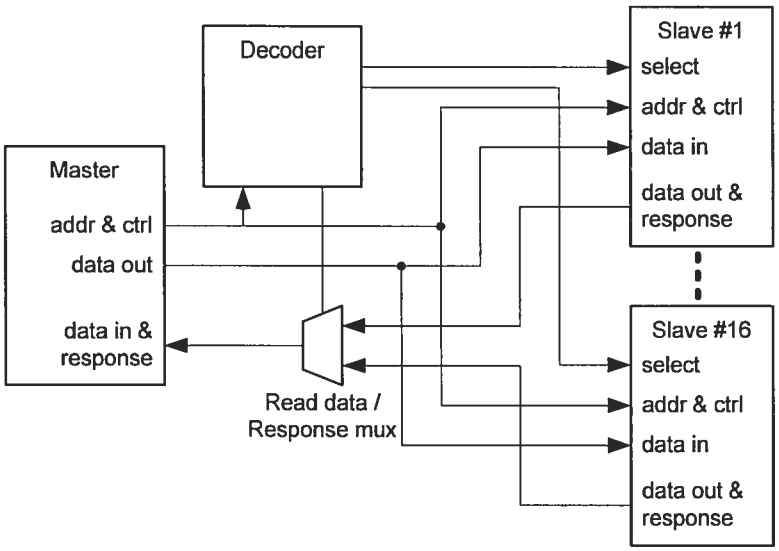

Figure 15 : Représentation schématique d'un bus AHB-lite.[3]

[3] Le dessin vient du mémoire [40]

Le bus AHB-Lite a trois composants principaux : un maître, des esclaves et un décodeur. Les maîtres et les esclaves sont en fait des modules externes représentés par des interfaces interagissant avec le bus. L'interface de type maître fait des requêtes de transfert. L'interface esclave reçoit et répond à des requêtes de transfert. Si un module doit initier et recevoir des requêtes de transfert, il doit implémenter les 2 types d'interface. À partir d'ici, on appelle un maître, l'interface maître d'un module, et un esclave, l'interface esclave d'un module. Les signaux utilisés par le bus sont présentés dans le Tableau 2. Les Figure 16, Figure 17 et Figure 18 présentent les interfaces des modules du décodeur, des esclaves et du maître.

Tableau 2 : Signaux du bus AHB-Lite

Nom du signal	**Définition**	**Source**
HCLK	Horloge du bus	Source de l'horloge
HRESETn	Remise à zéro	Contrôleur de remise à zéro
HADDR[31:0]	Bus d'adressage	Maître
HTRANS[1:0]	Type de transfert	Maître
HWRITE	Direction du transfert	Maître
HSIZE[2:0]	Taille du transfert	Maître
HBURST[2:0]	Type de rafale	Maître
HPROT[3:0]	Contrôle de protection	Maître
HWDATA[31:0]	Bus d'écriture	Maître
HSELx	Sélecteur d'esclave	Décodeur
HRDATA[31:0]	Bus de lecture	Esclave
HREADY	Fin de transfert	Esclave
HRESP[1:0]	Réponse au transfert	Esclave

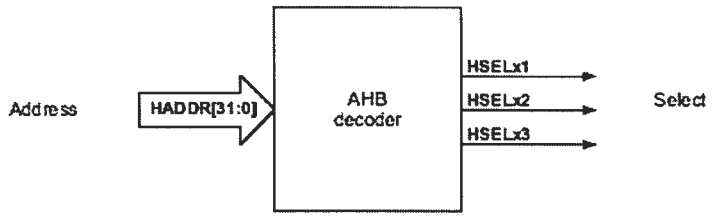

Figure 16 : Interface du décodeur du bus AHB-Lite[4]

[4] Image provenant de [58]

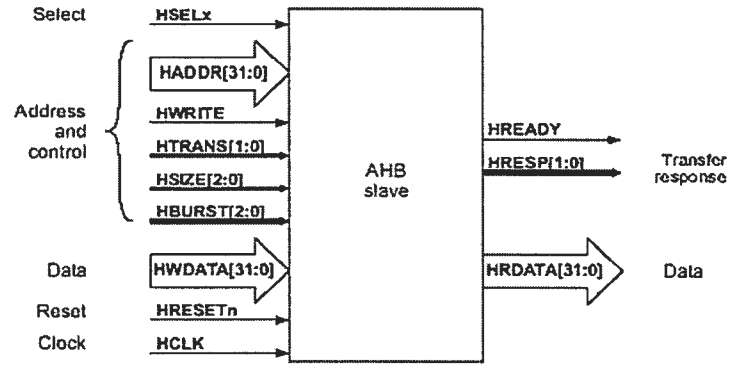

Figure 17 : Interface de l'esclave du bus AHB-Lite[5]

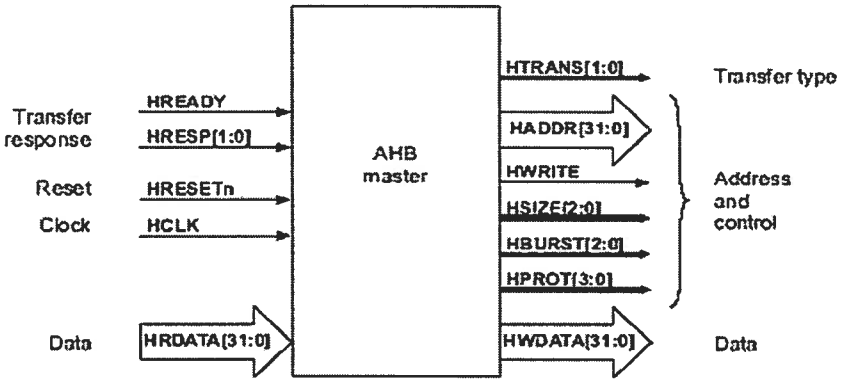

Figure 18 : Interface du maître du bus AHB-Lite[6]

Ce bus simplifié impose plusieurs contraintes au bus AHB. Il ne permet qu'un seul maître. Les esclaves ne peuvent donner qu'une réponse positive (OKAY) ou une réponse d'erreur (ERROR) à une requête du maître. Donc, ils ne peuvent pas demander au maître de réessayer la requête (RETRY) ou de diviser la requête en deux parties (SPLIT) comme dans le bus complet. Les esclaves peuvent toujours donner une réponse OKAY avec le signal d'attente (WAIT) pour avoir plus de temps avant de répondre à la requête. Comme il n'y a qu'un maître, il n'y a pas d'arbitre qui gère l'accès au bus

[5] ibidem

[6] ibidem

par les maîtres.

5.1.1. Exemple

Comme dit précédemment, le bus est une pipeline à deux étages : dans le premier cycle, le maître envoie les signaux d'adresse et de contrôle. Dans le cycle suivant, le maître envoie la donnée si c'est une écriture ou l'esclave envoie la donnée si c'est une lecture (Figure 19).

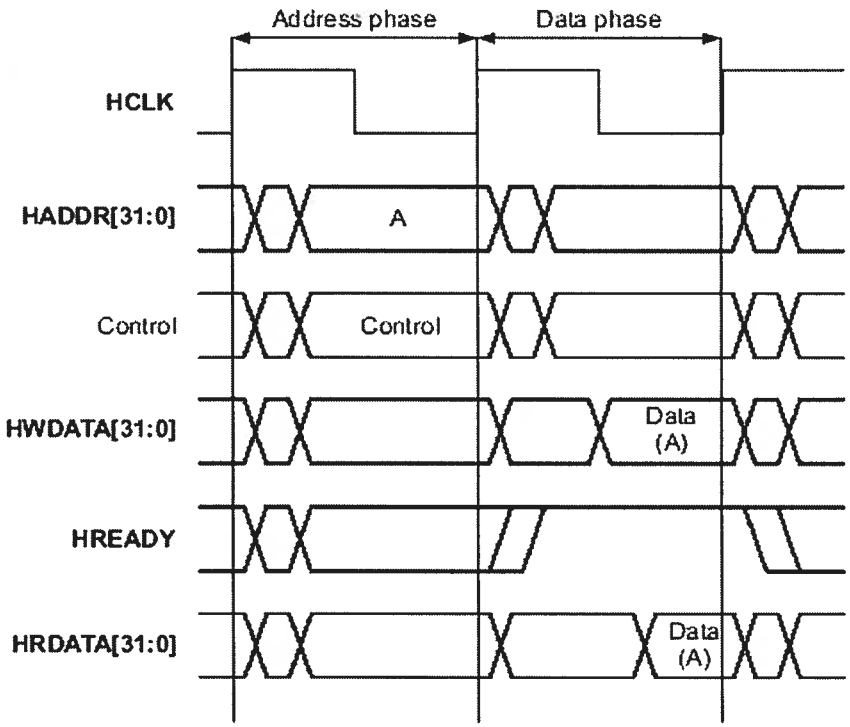

Figure 19 : Transfert simple du bus AHB.[7]

La Figure 20 montre le fonctionnement de la pipeline du bus. Dans le premier cycle, le maître envoie les signaux d'adresse et de contrôle sur le bus pour le transfert *A*. Mais dans le deuxième cycle, le maître envoie la donnée du transfert *A* en même temps qu'il envoie les signaux d'adresse et de contrôle du transfert *B*. Donc, dans le deuxième cycle, on a les transferts A et B qui sont exécutés en

[7] ibidem

parallèle. Dans le troisième cycle, l'esclave met le signal HREADY à 0. Quand ce signal est à 0, cela veut dire que l'esclave n'est pas prêt à répondre au transfert reçu durant le cycle précédent (*B*). Cela insère un cycle d'attente dans la pipeline et étire le cycle pour le transfert des signaux de contrôle et de l'adresse du transfert suivant (*C*). Dans le 4^{e} cycle, l'esclave reçoit le transfert *C* et donne la réponse du dernier transfert qu'il a fait attendre.

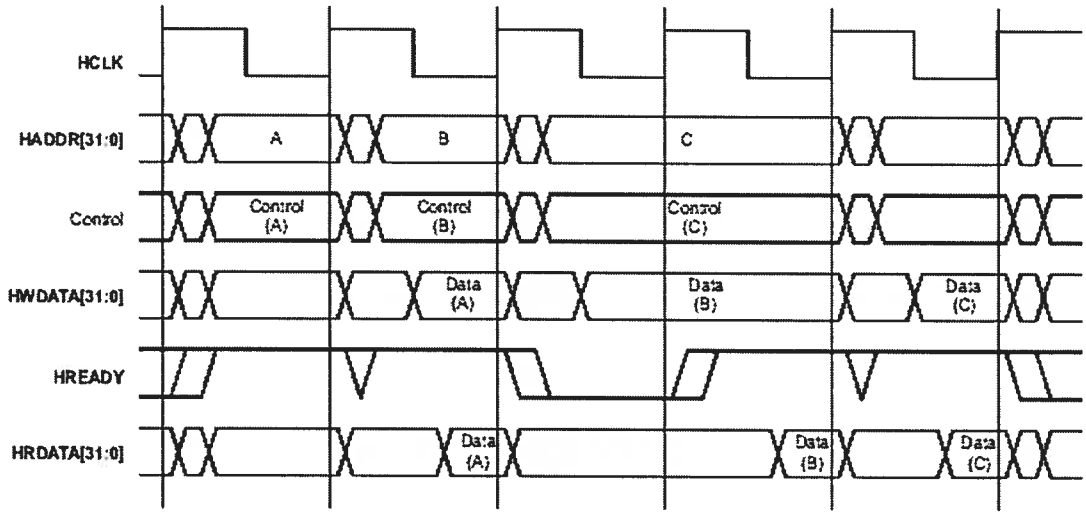

Figure 20 : Transfert multiple avec un état d'attente.[8]

Dans la Figure 21, on voit les signaux de contrôle : HTRANS, HBURST, HWRITE, HSIZE, HPROT. HTRANS est le type de transfert, il prend la valeur NONSEQ au début d'un transfert en rafale et SEQ par la suite. Si le maître ne veut pas faire de transfert, HTRANS prend la valeur IDLE et s'il veut insérer un délai dans le transfert il prend la valeur BUSY.

Le signal HWRITE est à 1 si le transfert du maître est une écriture et à 0 si c'est une lecture. Le signal HPROT est pour un système de protection de la mémoire. Ce n'est pas implémenté dans l'étude de cas. HSIZE est la taille de la donnée transférée à chaque cycle. Dernièrement, HBURST donne le type de transfert SINGLE est un transfert simple, INCR4, INCR8, INCR16 sont des transferts en rafale incrémentale de 4, 8 ou 16 transferts simples. INCR est un transfert en rafale

[8] ibidem

incrémentale, mais sans taille prédéterminée. On détecte la fin de ce type de transfert quand le signal HTRANS n'est plus SEQ ou BUSY. WRAP4, WRAP8, WRAP16 sont aussi des transferts en rafale, mais ils sont bouclant sur leurs adresses. Cela veut dire que si l'adresse initiale du transfert n'est pas alignée au nombre total d'octets envoyés dans la rafale (size x nb rafale), alors les adresses de la rafale vont boucler lorsque la limite va être atteinte. Par exemple, dans un transfert WRAP4 de 32 bits, les adresses bouclent à des bornes de 16 octets. Donc, si un transfert commence avec l'adresse 0x34, les adresses suivantes vont être 0x38, 0x3C et 0x30.

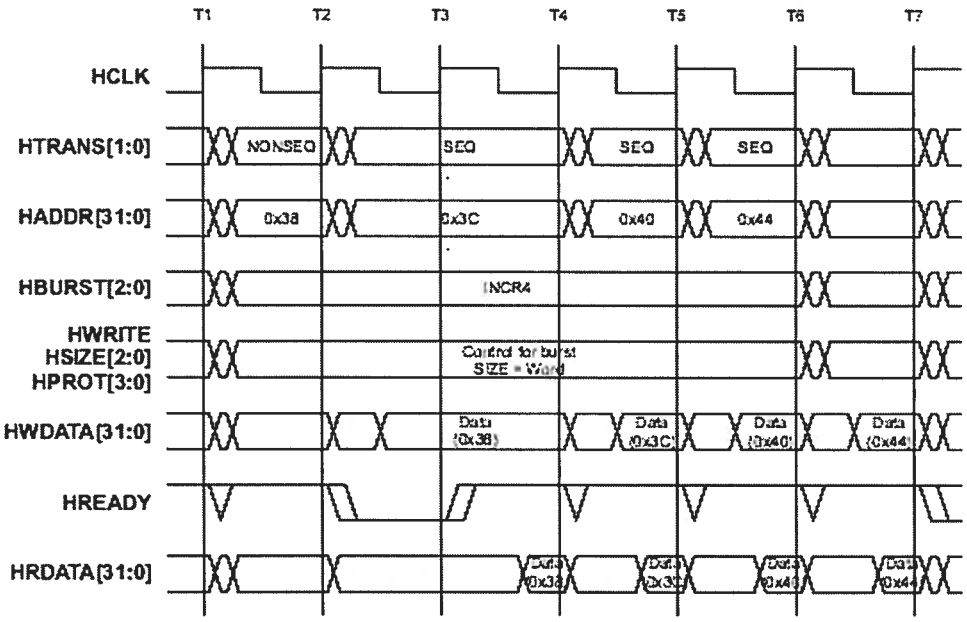

Figure 21 : Transfert avec le détail des signaux[9]

5.2. Études préalables

Des études préalables ont déjà vérifié formellement la spécification AHB d'AMBA. Par contre, elles ne couvrent pas tout ce qui a été vérifié dans ce mémoire. L'étude de Schmaltz-Borrione [61] utlise un outil d'aide à la preuve appelé ACL2 [62]. Leur modèle n'inclut pas toutes les

[9] ibidem

fonctionnalités de la spécification, dont la transmission des réponses de l'esclave soient *wait*, *split*, *retry* et *error*. Il ne modélise pas non plus les transactions en rafale, la pipeline de transfert et l'esclave par défaut. Le modèle fait l'hypothèse d'un algorithme d'arbitrage de type *round-robin*. Ceci est une des implémentations possibles de l'arbitre. Leur modèle ne considère donc pas les autres types d'arbitrage possibles. La spécification est donc partiellement couverte par leur modèle. Par contre, ils prennent soin de faire l'hypothèse d'un nombre arbitraire de maîtres et d'esclaves dans le cas de la vérification de six des propriétés de la spécification. Quant à la vérification des requis manquants, elle est incomplète.

Hasan Amjad [63, 64] a vérifié les bus AHB et APB de la spécification AMBA. Il a vérifié les bus individuellement par model checking en se basant sur la technique du μ calcul et le logiciel HOLCHECK [7]. Il a vérifié le système global des deux bus avec le vérificateur de théorème HOL [7]. Par contre, son modèle n'inclut que les types de transfert SINGLE et INCR4. Les mots non alignés sur la grandeur du bus ne sont pas vérifiés. Il s'agit pourtant là de cas d'erreurs potentiels. L'étude de Amjad procède à la vérification formelle de sept propriétés CTL mais ne vérifie pas la vivacité du système.

M. Newey a traduit la spécification du bus AMBA dans le langage Z [65]. Il ne fait pas de vérification formelle de propriétés. Mais comme le langage Z est un langage formel de description de spécification, cette version formelle devrait présenter l'avantage d'être sans ambiguïté. Il n'en demeure pas moins que le travail de Newey ne couvre pas la vérification de toutes les propriétés spécifiques du bus tel que nous le faisons dans ce mémoire.

Roychoudhury, Mitra et Karri [66] ont utilisé des techniques de vérification formelle sur le bus AHB d'AMBA pour vérifier des invariants de la spécification. Ils ont découvert un cas possible d'erreur, celui où un maître peut toujours se faire refuser l'accès au bus. Ils ont utilisé le model checker SMV de Cadence. Notre étude de cas ne couvre pas les transferts séparés (split) qui sont responsables de ce problème. Ce cas d'erreur ne concernera donc pas notre modèle.

5.3. Modélisation du bus AHB-Lite

Dans l'outil CPN tools, notre modèle se déploie sur une page supérieure appelée Top (Figure 22). Elle contient une page *Master* (Maître) et une page *Slave* (Esclave). La page Top décrit les transitions

effectuées par l'horloge et le décodeur. Les transitions effectuées par le maître et les esclaves ont été spécifiées de la façon la plus restrictive possible au moyen de préconditions suffisamment fortes. Ceci est important si l'on veut éviter le déclenchement de ces transitions dans des contextes où la spécification ne le commanderait pas explicitement. Ceci est nécessaire pour permettre la détection des requis manquants. Si les préconditions ne sont pas assez fortes, les transitions correspondantes pourront être déclenchées bien que la spécification ne l'autorise pas nécessairement, Les requis manquants, généralement détectés par une absence de comportement possible (un blocage) associé à un état donné du système, seront alors masqués par l'exécution de ces transitions plus laxistes que la spécification ne le sous-entend. Par conséquent, pour avoir un bon modèle, il est important de définir des préconditions qui correspondent précisément (et seulement) aux cas couverts par la spécification.

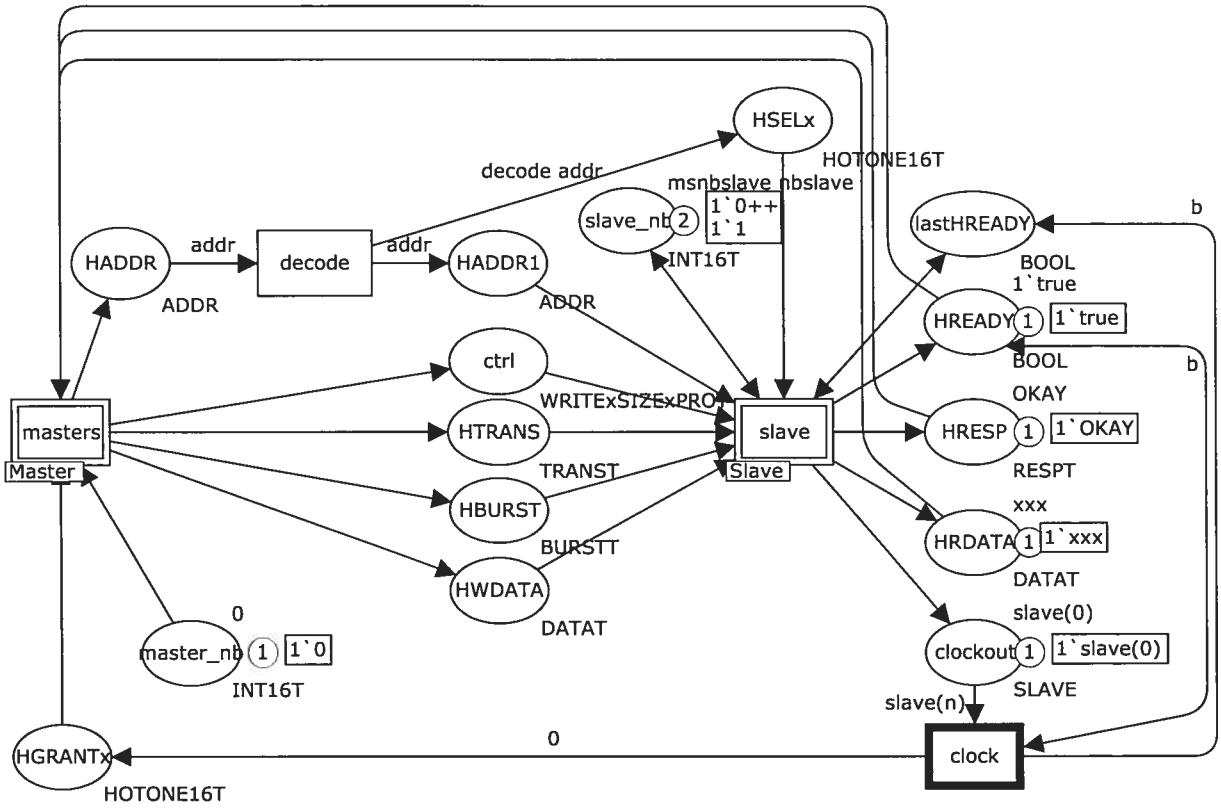

Figure 22 : Page racine de l'étude de cas

5.3.1. Démonstration

Dans cette section, un modèle simplifié de l'étude de cas sur AHB-Lite est présenté. Ce modèle

contient un maître et un esclave et ne permet que des transferts de lecture de type SINGLE, sans permettre que le maître envoie des signaux IDLE, BUSY et sans permettre à l'esclave d'envoyer des signaux WAIT.

Comme dans la méthodologie, on a commencé par créer les couleurs. Dans la Figure 18, on voit que le maître reçoit les signaux HREADY, HRESP et HRDATA. Les signaux HRESETn et HCLK ne sont pas modélisés, car le premier n'est pas implémenté et le deuxième est détecté par l'arriver des jetons. Il a comme signaux de sortie HTRANS, HADDR, HWRITE, HSIZE, HBURST, HPROT, HWDATA. On crée donc les couleurs suivantes : RESPT, DATAT, TRANST, ADDR, SIZET, BURSTT, PROTT. Pour les signaux HWRITE et HREADY, on utilise la couleur existante BOOL, car c'est leur type. Comme les signaux HWRITE, HSIZE et HPROT sont regroupés sous le nom de signaux de contrôle, on a créé une autre couleur CTRL qui est un tuplet contenant ces trois signaux. Si on regarde la Figure 17, l'esclave reçoit le signal HSELx qui provient du décodeur. On crée la couleur HOTONE16T qui contient le numéro de l'esclave qui est sélectionné. S'il n'y en a aucun de sélectionner, il contient -1 (~1 en SML). On garde ce signal même s'il n'y a qu'un esclave, car dans la version complète il peut y en avoir plusieurs. La couleur TRAN contient l'information que le maître a besoin de garder entre les cycles d'horloge. Voici la définition de la constante (val ...) ainsi que les définitions des couleurs (colset ...) :

```
val block = 64;
colset BOOL = bool;
colset E = with e;
colset ADDR = int with 0..(block-1);
colset DATAT = with data|xxx;
colset HOTONE16T = int with ~1..15;
colset TRANST = with IDLE|BUSY|NONSEQ|SEQ;
colset BURSTT = with SINGLE|INCR|WRAP4|WRAP8|WRAP16|INCR4|INCR8|incr16;
colset PROTT = with DATAF;
colset SIZET = with s8,s16,s32,s64,s128,s256,s512,s1024;
colset RESPT = with OKAY|ERROR;
colset CTRL = product BOOL*SIZET*PROTT;
colset TRAN = product INT*ADDR*BOOL*SIZET*PROTT*BURSTT;
```

Dans la méthodologie, on continue en créant une variable par couleur et on crée des variables

supplémentaires au besoin. Voici la déclaration de toutes les variables nécessaires dans cet exemple avec leur couleur respective :

```
var slav,n: INT;
var dat, dat2: DATAT;
var burst: BURSTT;
var prot: PROTT;
var addr, addr2: ADDR;
var size1, size2: SIZET;
var write: BOOL;
```

Ensuite, on crée la structure et l'interface des modules. Le module du décodeur a été inclus directement dans la page supérieure, car il contient une seule transition. Le module de l'esclave contient une seule transition et a sa propre page, car d'autres transitions sont ajoutées dans la version complète du modèle. Enfin, on ajoute le comportement dans les pages des modules. Le comportement des modules a été pris de la Figure 21. Voici les trois pages résultantes de l'exemple :

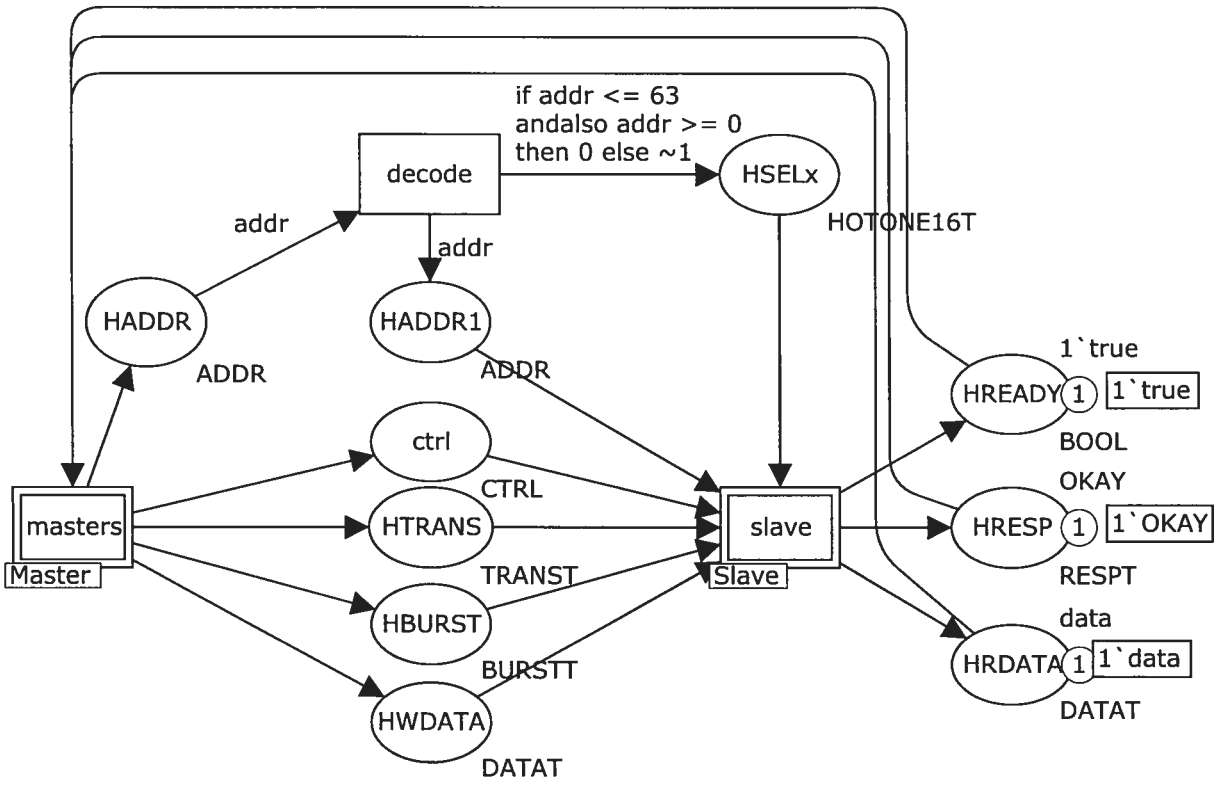

Figure 23 : Démonstration de l'étude de cas : page supérieure

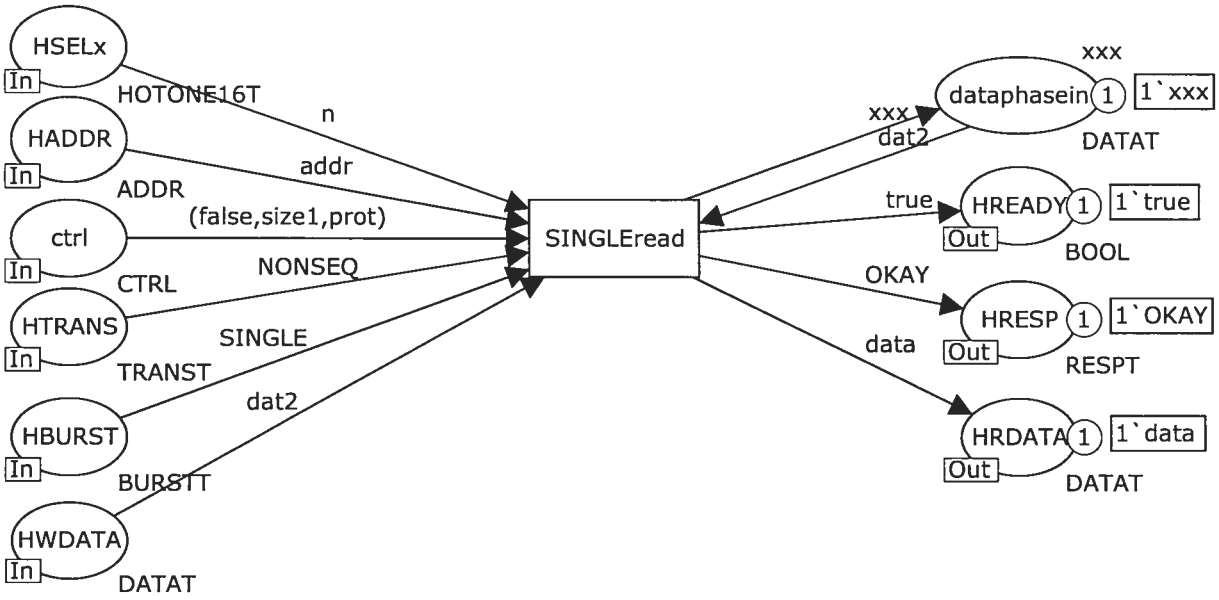

Figure 24 : Démonstration de l'étude de cas : page de l'esclave

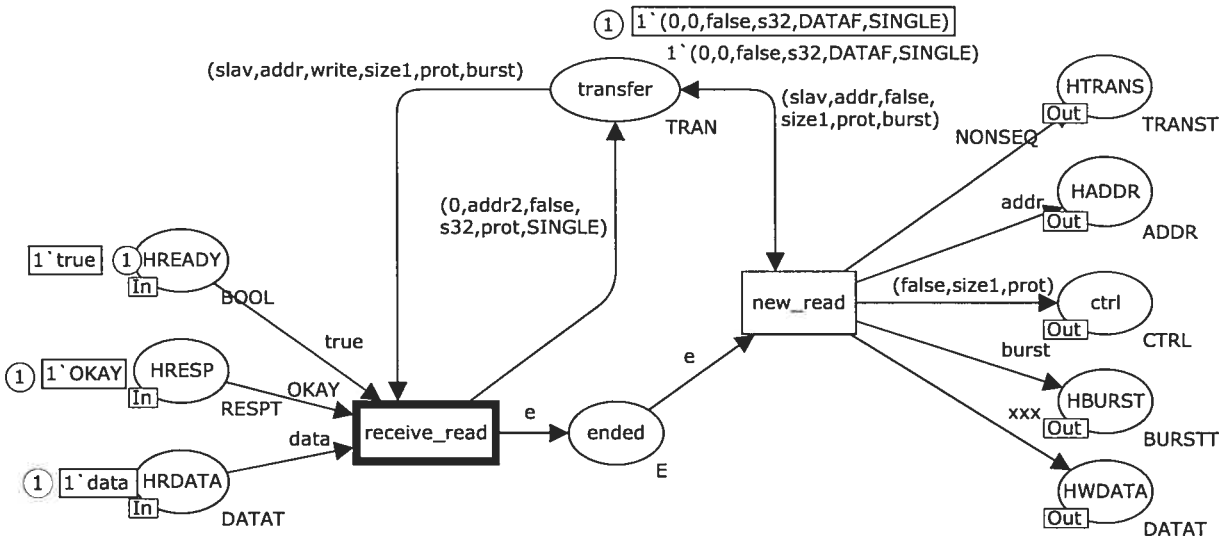

Figure 25 : Démonstration de l'étude de cas: page du maître

Le système est initialisé de telle sorte que le maître reçoive la réponse d'une ancienne requête. De cette façon, le modèle est plus simple, car on ne modélise pas l'initialisation du système. Le maître a deux transitions. La première, *receive_read*, reçoit la réponse de l'esclave si elle est valide. De plus,

cette transition génère aléatoirement le prochain transfert à effectuer. C'est la place *transfer* qui contient l'information dont le maître a besoin sur la requête en cours. Dans ce cas, comme toutes les requêtes sont de type SINGLE et qu'il n'y a pas de WAIT, on génère toujours une nouvelle requête. Dans la version complète, il faut générer une nouvelle requête seulement si le transfert a été fait au complet. La requête est générée par le tuplet *(0, addr2, false, s32, prot, SINGLE)*. Comme on peut voir la requête envoyée est celle d'un transfert de type SINGLE, de taille 32 bits à destination de l'esclave 0. La seule variable qui n'est pas liée est *addr2*. Comme elle n'est pas liée, l'adresse de destination est générée aléatoirement.

Après que le maître ait accepté la réponse de l'esclave et généré le prochain transfert, il envoie la requête de transfert avec la transition *new_read*. Ensuite, c'est le décodeur qui sélectionne l'esclave qui doit répondre à cette requête. Enfin, c'est l'esclave qui répond à cette requête. Comme le modèle est cyclique, on recommence.

La prochaine section parle des optimisations utilisées dans l'étude de cas.

5.4. Optimisation pour la vérification formelle

Il existe plusieurs algorithmes automatiques de réduction de l'espace d'états. L'un d'eux est basé sur la symétrie et a été testé sur les réseaux de Petri colorés (Condensed State Spaces for Symmetrical Coloured Petri Nets) [22]. Par contre, l'outil CPN Tool n'en implémente aucun. Nous avons donc misé sur l'application de techniques d'abstraction, qui sont proposées dans la méthodologie, que nous appliquons durant la conception du modèle en vue de limiter l'espace d'états et le nombre de places.

5.4.1. Réduction de l'espace d'état

Pendant la conception, notre méthodologie propose la définition de certaines abstractions bien choisies pour limiter l'espace d'états. Nous décrivons ci-après les principales abstractions que nous avons définies pour notre étude de cas.

Les compteurs constituent des sources bien connues d'explosion exponentielle du nombre d'états. Ainsi, au lieu d'implémenter des compteurs pour cumuler le nombre de réponses successives *wait* et *busy* qu'un processus reçoit, le modèle mémorise seulement le signal (*wait* ou *busy*) associé au dernier transfert. Cela a pour conséquence qu'il est impossible de forcer le maître de faire un nombre

maximal de *busy* et c'est la même chose avec l'esclave avec le signal *wait*. Donc dans le modèle on a des cas limites ou le maître fait infiniment des *busy* et les esclaves infiniment des *wait*.

Dans la spécification, la plage d'adresse est séparée en blocs qui doivent être un multiple de 1ko pour s'assurer qu'une requête n'affecte qu'un seul esclave. Chaque esclave a comme plage d'adresses un nombre de blocs successifs. De plus, chaque requête en rafale du maître doit pouvoir être stockée dans un seul bloc. Si une requête nécessite plusieurs blocs, le maître doit la séparer en plusieurs requêtes. Ceci est important afin d'éviter qu'une requête ne soit répartie sur plusieurs esclaves, car la spécification ne le supporte pas. Pour la vérification formelle, il n'est pas essentiel de maintenir des blocs aussi gros, car les adresses du milieu du bloc ont une relation d'équivalence avec seulement les adresses qui changent. C'est-à-dire que leur traitement est identique sauf pour les numéros d'adresses qui sont modifiées de manière correspondante. On a donc utilisé des blocs de 64 octets, car certains types de transfert (INCR8 et WRAP8) échangent des données 8 fois et que le bus du modèle est de 32 bits. Cela donne un total de 32 octets par transfert. Pour être sûr de ne pas ajouter des cas qui n'existent pas dans la spécification, on ne prend pas des blocs de la même taille. Donc, on a pris le multiple de deux suivant. Cette taille de bloc réduit de manière exponentielle la taille de l'espace d'états et n'affecte pas la vérification à effectuer. Dans le modèle, la taille du bloc est paramétrable et on peut donc l'ajuster si nous voulons faire la vérification en considérant tout l'espace d'états.

En plus de réduire la taille du bloc d'adresses, par des classes d'équivalence, on ne permet que certaines adresses du bloc pour le départ des transferts. On permet seulement l'adresse 0 et l'adresse à la fin du bloc qui est le cas limite pour le type de transfert en cours. On peut faire cela, car les adresses entre ces deux adresses sont dans la même classe d'équivalence que l'adresse 0. Tous les autres cas entre ces deux adresses sont dans la même classe d'équivalence que l'adresse de début pour les transferts de type INCR, INCR4, INCR8, INCR16. Pour les transferts de type WRAP4, WRAP8 et WRAP16, certaines adresses sont équivalents à l'adresse limite de fin.

Au lieu de représenter les valeurs des données envoyées, on se contente de modéliser le fait qu'une donnée soit envoyée ou non. Pour ce faire, on utilise 2 symboles : *data* et *xxx*. Le symbole *data* représente le fait qu'on envoie une donnée et le symbole *xxx* le fait qu'on n'en envoie pas. Ceci est suffisant puisque le comportement du système (qu'on souhaite vérifié) est indépendant de la donnée envoyée.

Comme le modèle supporte un nombre variable d'esclaves, si tous les esclaves s'exécutent à chaque cycle, il y alors beaucoup d'états intermédiaires à considérer étant donnée l'entrelacement de leur exécution. Une solution triviale consisterait à séquentialiser leur exécution. On peut pousser davantage l'optimisation puisqu'on constate que la majorité des esclaves sont inactifs pendant un cycle. En effet, puisque qu'une seule requête est traitée par cycle, seul le maître et un des esclaves sont actifs pendant ce cycle. On peut donc considérer que l'exécution des esclaves qui sont dans les deux étages de la pipeline, c'est-à-dire, l'esclave qui répond à une requête précédente et l'esclave qui reçoit une nouvelle requête. Ceci est souvent le même esclave, mais dans des cas limites ce n'est pas vrai.

Pour éviter de créer des états supplémentaires avec des combinaisons différentes, on se limite à 1 jeton par état. Si une place n'a pas besoin de jeton, on crée une transition qui l'enlève plutôt que de le garder pour rien. Cela peut aider à diminuer le nombre d'états en évitant que des jetons inutiles soient la seule différence entre deux états.

On a séquentialisé l'exécution entre le maître et les esclaves pour diminuer le nombre d'états intermédiaires inutiles dû à l'entrelacement entre l'exécution du maître et de l'esclave. En plus, cela évite de créer des processus parallèles dans l'esclave et diminue davantage le nombre d'états intermédiaires.

5.4.2. Réduction du nombre de places

Pour diminuer le temps de vérification, on a réduit le nombre de places et donc le nombre d'états à explorer.

D'une part, on a constaté qu'on pouvait réduire le nombre de places simplement en corrigeant la duplication des places due à la décomposition du réseau de Petri en différent niveaux hiérarchiques. Les places qui ont la même signification (p. ex. représente une même variable), mais qui se trouvent sur des niveaux différents, peuvent être fusionnées en une seule place que l'on conserve dans le niveau hiérarchique supérieur. Les valeurs contenues dans cette place peuvent alors être utilisées par les niveaux inférieurs via des places de paramètres. De cette façon, on a une seule instance de la place commune aux instances inférieures.

D'autre part, on peut également réduire le nombre de places internes du modèle en modélisant le

traitement d'un module par un minimum de transitions et donc par un minimum de places pour stocker les résultats intermédiaires. Dans le meilleur des cas, on aurait recours à seule une transition par module et on n'aurait besoin d'aucune place intermédiaire entre la lecture des jetons d'entrée et la génération des jetons de sortie des modules (Figure 26a). Cette approche présente toutefois l'inconvénient de complexifier la modélisation en associant des fonctions complexes aux transitions. Nous avons opté pour une solution moins draconienne : le traitement des modules est avec un seul état intermédiaire (modélisé par des places) (Figure 26b). On limite ainsi la taille du modèle à vérifier.

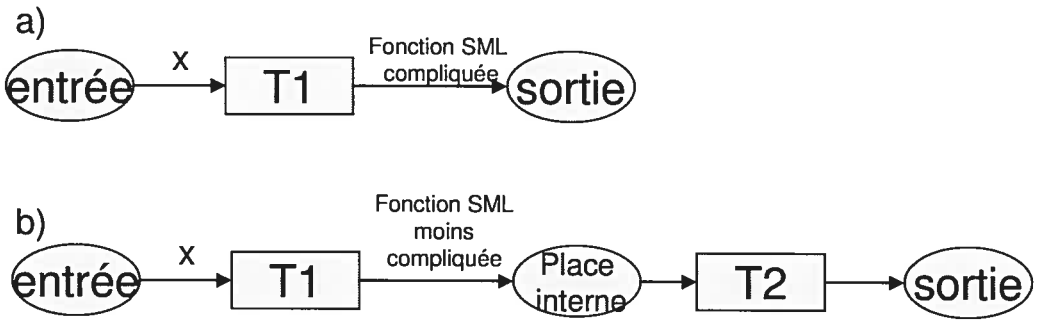

Figure 26 : Module de réseaux de Petri avec a) 0 état intermédiaire et b) 1 état intermédiaire

5.5. Vérification formelle

La vérification formelle du modèle de notre étude de cas est réalisée de façon semi-automatique grâce à l'outil CPN Tools. Pour ce faire, on utilise les fonctionnalités d'analyse du graphe d'états et d'analyse du graphe de composante fortement connexe offertes par l'outil [22, 67]. Malheureusement, CPN Tools ne dispose pas encore de générateur de graphe condensé ni de vérificateur de propriétés LTL. Il offre néanmoins un vérificateur de propriétés CTL et propose certaines optimisations, pour un sous-ensemble de CTL, en se basant sur le graphe de composantes fortement connexes. Contrairement aux model-checkers usuels, il ne fournit pas de contre-exemples pour les propriétés invalidées. Donc nous ne l'avons pas utilisé.

Nous avons opté pour une vérification incrémentale du modèle : une analyse est lancée à chaque incrément de la modélisation. Chaque analyse nécessite une série d'étapes d'opérations automatiques et manuelles. Les opérations suivantes sont d'abord exécutées séquentiellement par l'outil : génération du code de l'espace d'état, génération du graphe d'états accessibles, génération du graphe des composantes fortement connexes, génération du rapport d'analyse. On doit ensuite procéder

manuellement à la révision du rapport d'analyse pour s'assurer de la complétude du modèle et de certaines propriétés de cohérence. La vérification formelle des propriétés supplémentaires est finalement effectuées en parcourant le graphe d'états.

CPN tools permet de stopper la génération de l'espace d'états selon différents critères dont l'un deux est le temps de calcul. Par défaut, l'outil interrompt le calcul du graphe d'états après 5 minutes. Il importe donc de s'assurer que le graphe d'états a été généré au complet avant de poursuivre avec l'analyse. Cette information se trouve dans le rapport d'analyse. On peut également l'obtenir automatiquement en exécutant la fonction prédéfinie `EntireGraphCalculated()`.

5.5.1. Vérification de la finitude

On dit qu'un réseau est k-borné si chacune de ses places compte au plus k jetons. Le cas échéant, son graphe d'états est nécessairement fini. Cette propriété est vraie pour les réseaux de Petri Place/Transition (P/T). Comme tout réseau de Petri coloré peut être déplié en réseau P/T, cette propriété est donc également valable pour les réseaux colorés. On vérifie généralement cette propriété par une analyse structurelle du réseau (c.-à-d. identification d'invariants de places et de transitions), évitant ainsi de construire le graphe d'états complet, possiblement très grand. Malheureusement CPN tools n'offre pas de fonctionnalités permettant l'analyse structurelle du modèle. Nous avons donc choisi de limiter le temps de la génération du graphe d'états. Au-delà d'une certaine durée prédéterminée, nous stoppons la génération. Si le graphe d'états a été complètement construit pendant ce temps, c'est qu'il est fini. Autrement, on considère qu'il est soit infini, soit très grand. S'il est très grand ou infini, il faut modifier le modèle par abstraction ou autrement. La durée utilisée est approximative. Par défaut, le logiciel arrête après 5 minutes. Comme nous avons gardé la durée de génération du graphe d'état entre 1 et 2 minutes s'il n'y a pas de problème, on a utilisé le temps par défaut de 5 minutes. Par contre, ce qui peut-être fait, c'est de se baser sur le temps pris lors de la dernière génération du graphe et de l'augmenter si nos modifications devraient augmenter la taille du graphe d'états.

5.5.2. Vérification de la complétude

Chaque état terminal détecté dans le graphe d'états du modèle est susceptible de correspondre à une opération du système (en l'occurrence le bus AMBA) dont on a omis de spécifier un des

comportements. Pour chaque état terminal détecté, il importe donc d'identifier la cause de cette terminaison : erreur de modélisation ou incomplétude de la spécification modélisée. L'analyse des états terminaux nous a permis de relever plusieurs points de la spécification qui portent à confusion. La raison de cette confusion est le fait que le comportement nécessaire n'est pas exprimé clairement et explicitement. Il a fallu une analyse plus poussée de la spécification pour comprendre le comportement souhaité, car notre première interprétation de la spécification était erronée. Nous résumons les résultats de cette analyse dans la section 5.6. À noter que, dans le cas du bus AMBA, la vérification nous a permis d'identifier essentiellement des imprécisions, mais aucun réel requis manquant.

La liste des états terminaux peut être consultée dans le rapport (produit par CPN tools) faisant état de l'analyse du graphe de composantes fortement connexes. Si cette liste d'états terminaux est grande, il peut être fort utile d'en optimiser l'examen. Pour ce faire, on trie les états terminaux selon la taille croissante des séquences de transitions qui les relient à l'état initial. En effet, les états terminaux accessibles plus tardivement sont souvent une variante d'une erreur elle-même à l'origine d'un état terminal obtenu plus tôt. Comprendre les causes d'un certain état terminal permet souvent de comprendre celles des états terminaux successifs. Puisque CPN tools génére le graphe d'états en largeur (et non en profondeur), sa numérotation des états permet facilement d'identifier ceux qui sont plus près de l'état initial. On parcourt donc les états terminaux dans l'ordre croissant du numéro qui leur est attribué dans le graphe d'états. Pour trouver l'état terminal le plus près de l'état initial, on crée une fonction SML qui prend une liste de numéro d'états et retourne le numéro le plus petit :

```
fun minlist [head]=head |
    minlist (head::tail) = let val min=minlist tail in
                               if head<min then head else min end;
```

Donc, pour déterminer le prochain état terminal à analyser (avec le minimum de transition pour l'atteindre), on exécute la commande SML : `minlist ListDeadMarkings()`. La fonction prédéfinie `ListDeadMarkings()` retourne la liste des numéros des états terminaux.

Comme dit précédemment, il n'y a aucun requis manquant qui a été trouvé.

5.5.3. Vérification de la cohérence

On vérifie les propriétés de cohérences à la section 4.3.4 pour garantir la cohérence du modèle. Se référer à cette section pour l'explication des propriétés. Pour vérifier ces propriétés, nous étudions d'abord les résultats décrits dans le rapport d'analyse produit automatiquement par CPN tools. Le rapport nous informe du nombre d'états composant les graphes, de la vivacité des transitions, de la présence de transitions mortes et de la quantité minimale et maximale de jetons dans les places. Pour compléter la vérification, nous devons examiner plus en détail les graphes produits par CPN tools, ce que nous faisons en définissant de nouvelles fonctions d'analyse de ces graphes.

Un des risques du model-checking est d'introduire des erreurs de modélisation, tout à fait indépendantes de la cohérence du système de requis à vérifier. Pour tenter de limiter les erreurs qu'il pourrait introduire malgré lui, le développeur du modèle peut faire quelque vérification à chaque ajout qu'il fait au modèle. Par exemple, chaque fois qu'il modifie le modèle, le développeur sait si le nombre d'états doit augmenter ou diminuer (on a ajouté des fonctionnalités ou on a fait des optimisations). Si ce n'est pas ce qui est observé dans le rapport, c'est un indice qu'une erreur de modélisation a probablement été commise. On peut aussi utiliser le graphe de composantes fortement connexes. Ce graphe doit comporter beaucoup moins de nœuds que le graphe d'états, car dans notre cas le modèle est cyclique. Dans l'étude de cas, il y a 63 nœuds dans le graphe de composantes fortement connexes et 50 254 états dans le graphe d'états.

Ce qui nous intéresse dans notre modèle, ce sont les transitions non vivantes au lieu des transitions vivantes, car le modèle étant cyclique, beaucoup de transitions doivent être vivantes. Les seules qui ne devraient pas l'être sont celles qui peuvent être désactivées en permanence. On peut vérifier cela en comparant manuellement la liste des transitions vivantes dans le rapport et la liste de toutes les transitions du modèle. On peut l'automatiser en utilisant des fonctions prédéfinies : `TI.All - ListLiveTIS()`. La structure ML `TI.All` donne la liste de toutes les transitions du modèle et `ListLiveTIs()` donne la liste des transitions vivantes. La fonction - est une fonction sur des ensembles multiples qui enlève le contenu du deuxième ensemble multiple au premier. Dans notre modèle, il y a une seule transition non vivante : *TI.Master'chose_request 1*. Cela ne pose pas de problème de cohérence, car c'est une transition d'initialisation.

Ensuite, on vérifie qu'il n'y a pas de transitions mortes, car cela n'est pas la même chose que les

transitions non vivantes. Dans l'étude de cas, il n'y en a pas. S'il y en avait, cela indiquerait une erreur de modélisation.

Dans le rapport généré par l'outil, on peut vérifier le nombre maximum et minimum de jetons dans chaque place. Cela permet de vérifier que le modèle respecte la méthodologie de modélisation et qu'il n'y a pas d'erreur dans le modèle qui modifie la quantité de jetons dans les places. Dans notre modèle, il y a un maximum de 1 jeton dans chaque place sauf une qui contient des constantes. Cette place de constantes contient les numéros des esclaves existants dans le modèle : dans notre modèle il y a deux esclaves avec les numéros 0 et 1. Le minimum est de 0 pour les places d'interface entre les modules et de 1 ou 0 pour les places internes selon leur utilisation. La place constante a un minimum de 2 jetons.

Finalement, le rapport permet de vérifier les propriétés d'équité des transitions individuelles. Cela sert à vérifier si l'ordre d'exécution des transitions est vraisemblable. Par exemple, si le bus AMBA n'est pas équitable, on pourrait avoir le cas ou le maître ou un esclave n'est jamais exécuté et cela n'est pas souhaitable. Les transitions de décodage de l'adresse et d'horloge sont bien impartiales, car elles sont obligatoires à chaque cycle. Toutes les autres transitions sont fortement équitables ou faiblement équitables. On définit une fonction `ListNoFairnessTIs()` pour identifier directement les transitions ne respectant aucune forme d'équité. Les propriétés faiblement et fortement équitables ont été vérifiées pour chacune des transitions. Elles ont toutes la bonne propriété. (Voir chapitre 4 pour explication)

```
fun ListNoFairnessTIs() = List.foldl (fn (x,l)=>(List.filter (fn n=>
(n=x)) l )) (TI.All) (ListJustTIs());
```

On peut vérifier manuellement d'autres propriétés d'équité. Si on n'oublie pas qu'une propriété d'équité peut être appliquée à un ensemble de transitions, on peut en vérifier une qui est intéressante : l'ensemble des transitions d'un module synchrone doit être impartial, ce qui est vrai dans notre étude de cas, car au moins une transition de l'ensemble doit être exécutée à chaque cycle. Pour vérifier cette propriété, on utilise la fonction existante : `TIsFairness` qui prend en entrée une liste de transitions et donne la propriété d'équité de cette liste. Pour l'ensemble des transitions dans les modules d'esclave et de maître, on a la propriété d'impartialité.

On peut vérifier les transitions qui sont concurrentes. Cela permet de vérifier s'il n'y a pas de transition qui ne doivent pas être concurrente ou qui le devrait et ne l'ai pas. On vérifie le nombre maximum de transitions activables en même temps. Dans le modèle, il y en a 2. On trouve cette

information avec la fonction suivante :

```
fun MaxTIEnabled () : int = SearchAllNodes(fn _ => true, fn n =>
length(remdupl(map ArcToTI(OutArcs n))),0,fn (x,y) => if x > y then x else
y);
```

Ensuite, on vérifie que les ensembles de transitions concurrentes trouvés sont autorisés par la spécification. Pour ce faire, on utilise une fonction qui génère la liste des ensembles de transitions concurrentes de taille 2. Dans l'étude de cas, elles sont toutes valides. Voici la fonction qui génère cette liste d'ensembles :

```
fun ListMaxTIEnabled () = SearchAllNodes(fn _ => true, fn n =>
(length(remdupl( map ArcToTI(OutArcs n))),[remdupl(ArcsToTI(OutArcs n))]),
(0,[]),fn ((x1,x2),(y1,y2)) => if x1 > y1 then (x1,x2) else if x1=y1 then
(x1,(append x2 y2)) else(y1,y2));
```

Parmi cette liste, il y a des ensembles qui donnent les transitions qui sont en conflit. Dans l'étude de cas, toutes les transitions concurrentes sont en conflit. Ceci est dû au fait que l'exécution parallèle du maître et des esclaves a été séquentialisée et qu'il n'y a pas de parallélisme à l'intérieur des modules. Donc, tous les conflits sont entre des ensembles de transitions qui sont dans le même module. Si on vérifie, toutes les transitions en conflits ont la propriété d'équité faible. En particulier les transitions *MasterEndRW'end 1* et *MasterEndRW'end 2* sont équitablement fortes, car même si elles sont en concurrence avec d'autres transitions, elles doivent toujours être activées, car elles ferment les transferts.

5.5.4. Vérification de propriétés supplémentaire

Nous avons reçu des propriétés supplémentaires sous forme de texte sur la spécification AHB. Un sous-ensemble couvre la spécification AHB-Lite et nous les avons utilisées. Ces propriétés supplémentaires peuvent être écrites en LTL. Cela a été fait pour l'étude de cas d'un vérificateur de propriété LTL fait par Michel Metzger pour des modèles écrit en ESys.Net [11, 68].

Comme CPN tools ne dispose que d'un vérificateur de propriétés ASK-CTL qui est une extension de CTL et qu'il ne donne pas de contre-exemple, on a opté pour une autre approche : l'utilisation directe du graphe d'états.

Faire la vérification par cette approche demande qu'on crée une structure supplémentaire pour parcourir le graphe d'états. Il existe déjà des fonctions de base pour parcourir le graphe, mais comme

on vérifie les propriétés supplémentaires aux fronts montants de l'horloge, on doit créer des fonctions qui naviguent entre ces états pour ne pas tenir compte des états intermédiaires. Ensuite, comme vérifier toutes ces propriétés manuellement à chaque modification du modèle peut prendre du temps, on a créé des fonctions qui le font automatiquement.

Pour naviguer entre les états au front de l'horloge, on doit connaître une condition sur un état qui dit si cet état est au front de l'horloge. Comme le modèle a une exécution séquentielle du maître et des esclaves, on a deux conditions : une avant que les esclaves s'exécutent (la seule transition activable est TI.Top'decode) et une avant que le maître s'exécute (la seule transition activable est TI.Top'clock). Pour faire les fonctions qui vérifient cette condition, on a besoin des fonctions OutArcs et ArcsToTI. La première donne une liste des arcs sortant d'un état et la seconde donne la liste des transitions correspondantes à la liste des arcs qu'elle reçoit. Voici ces deux fonctions :

```
fun isNodeMasterResponse nod = (ArcsToTI(OutArcs nod)=[TI.Top'decode
1]);
fun isNodeSlaveResponse nod = ArcsToTI(OutArcs nod)=[TI.Top'clock 1];
```

Avec ces fonctions, on crée d'autres fonctions qui prennent un état et donne la liste des états au prochain front d'horloge. Voici leur signature :

```
val FromMasterNextNodesSlaveResponse = fn : Node -> Node list
val FromSlaveNextNodesMasterResponse = fn : Node -> Node list
```

Avec les fonctions de navigation créées, on a pu vérifier toutes les propriétés supplémentaires qui s'appliquent au modèle et les propriétés qu'on a rajoutées. Elles sont toutes valides. Par exemple, la première règle dit qu'au front de l'horloge toutes les sorties du maître doivent être valides sauf pour la donnée. Comme l'outil vérifie que le type des jetons dans les places est valide, il faut vérifier que les places de sortie ne soient pas vides. Pour ce faire, on utilise la fonction *PredNodes* qui parcourt le graphe d'état et applique la fonction du deuxième paramètre à tous les états.

```
val q1=PredNodes(EntireGraph,fn n=> if(isNodeMasterResponse n)then not(
     Mark.Master'HADDR 1 n<>empty
     andalso Mark.Master'HTRANS 1 n<>empty
     andalso Mark.Master'ctrl 1 n<>empty
     andalso Mark.Master'HBURST 1 n<>empty) else false, 10);
```

Pour terminer, il y a quelques propriétés supplémentaires qui ont été ajoutées à la liste obtenue.

Entre autres, on vérifie que le modèle autorise tous les types de transfert possibles : de type simple (1 seul bloc), incrémental de 4, 8 ou 16 blocs, *bouclant* de 4, 8 ou 16 blocs ou incrémental d'un nombre indéterminé de blocs. Vérifier cette possibilité garantit qu'il n'y a pas de transition « type de transfert » qui a été désactivée par erreur ou volontairement pour diminuer la taille de l'espace d'états. Voici les fonctions utilisées :

```
PredNodes(EntireGraph,fn n=>Mark.Master'HBURST 1 n=1`x,1); pour tous x ϵ
(INCR4, INCR8, INCR16, WRAP4, WRAP8, WRAP16, SINGLE)
```

5.6. Résultats

Les résultats des vérifications faites pendant et après la modélisation du bus AHB-Lite sont en deux parties. On a trouvé des sections de la spécification qui sont ambiguës et on a trouvé parmi les règles qui nous ont été données des inconsistances et qu'il manquait des règles simples. Aucun requis manquant n'a été trouvé.

5.6.1. Problèmes trouvés dans la spécification

Il y a quelques actions qui ne sont pas claires dans la spécification. Il y a un signal HREADY que tous les esclaves donnent en sortie. L'esclave qui reçoit une requête de transfert peut mettre ce signal à bas pour prolonger cette requête. Pour chaque cycle que l'esclave met ce signal à bas, la requête est prolongée d'un cycle. Le maître reçoit la valeur multiplexée de HREADY des esclaves. C'est l'esclave du dernier transfert qui est sélectionné.

La spécification ne fait pas de différence entre les valeurs du signal HREADY qui sort des esclaves (HREADYx, ou x est le numéro de l'esclave) et celle qui est le résultat du multiplexeur de ces signaux (HREADYmul). Dans la section 3.8, il est dit que les esclaves doivent accepter une requête de transfert si le signal HSEL qu'il reçoit est haut et que le signal HREADY est haut. Le problème est qu'on ne sait pas quelle version du signal on prend, soit HREADYx ou HREADYmul. Dans le modèle de l'étude de cas, on a pris le signal HREADYx de l'esclave.

Cette ambiguïté cause un problème dans un cas. Ce n'est pas défini explicitement ce que le maître doit faire pour commencer un nouveau transfert si la valeur du signal HREADY reçu par le maître est bas pour signaler que l'esclave prolonge le dernier transfert. Soit le maître commence le transfert pour rien jusqu'à ce que l'esclave soit prêt ou il fait IDLE jusqu'à ce que l'esclave soit prêt. Si tous les

esclaves reçoivent le signal HREADYmul, ils peuvent faire la différence entre les deux et ne pas considérer le transfert. Par contre, d'après la Figure 17 qui vient de la spécification, ils ne reçoivent pas ce signal. Il est donc conseillé que le maître envoie comme type de transfert le type IDLE en attendant de recevoir le signal HREADY à haut, car sinon l'esclave ne sait pas s'il doit commencer ou non. L'autre option est que les esclaves reçoivent aussi la valeur du signal HREADY multiplexé.

Il y a un autre cas où cela pose problème quand l'esclave donne comme réponse une valeur basse au signal HREADY et que le maître envoie une requête avec comme valeur un IDLE ou un BUSY. Dans la section 3.5 de la spécification, l'esclave doit répondre à toutes requêtes de type IDLE ou BUSY, mais il est en train de traiter la dernière requête. Cela est résolu si le maître ne considère pas les requêtes qu'il fait, c'est-à-dire qu'il n'attend pas de réponse pour cette requête, quand il reçoit un signal HREADY bas.

Il n'y a pas d'exemple de plusieurs esclaves. Cela rend difficile la compréhension des cas limites ou deux requêtes successives ne sont pas destinées à des esclaves identiques.

Il n'y a pas d'indication où se trouvent les registres dans le système pour faire changer de cycles d'horloge. Trouver l'emplacement de ces registres n'est pas facile à faire à partir de la spécification. L'emplacement des registres dépend de chaque signal et cela est compliqué par l'architecture pipeline du bus.

Il manque des signaux dans plusieurs figures dont les figures de la spécification AMBA 3-2, 3-12, 3-19 et 3-35 ainsi que la figure 2 dans la spécification d'AHB-Lite. Ceci n'est pas très grave s'il y a aussi des graphiques clairs qui indiquent tout, ou qu'il est indiqué que le graphique est incomplet. Ce n'est pas le cas. Le module de décodeur dans ces figures n'a pas en entrée le signal HTRANS et n'a pas en sortie le signal HRESP qu'il a besoin pour l'esclave par défaut qu'il devrait implémenter (page 3-19). Mais le plus problématique, c'est que dans la section dédiée au décodeur, 3.21, il n'en parle pas du tout. Cette section devrait tout avoir sur le décodeur.

5.6.2. Problèmes trouvés dans les règles

En effectuant la vérification formelle des règles reçues, on a trouvé plusieurs problèmes par rapport aux règles. Certaines sont problématiques et il manque des règles pour vérifier toute la spécification.

Dans la section 3.6.1 de la spécification, il est dit que les esclaves doivent vérifier si le transfert en

rafale est annulé. Un transfert en rafale est annulé si le signal HTRANS indique IDLE ou NONSEQ avant qu'un esclave revoie la dernière requête du transfert. Il n'y a aucune règle qui vérifie cette propriété. Voici cette propriété formalisée en LTL, même si on n'utilise pas ce langage.

```
if(localburstleft>0 && (htrans==IDLE || htrans==NSEQ) && lasthready==true && localburst!=INCR)
```

Dans les règles, il n'y a aucune règle qui vérifie la couverture des cas possibles. Cela est important, car si par erreur on désactive des réponses de l'esclave ou certains types de transfert par le maître, on fait une vérification partielle des propriétés. Donc si le cas de la propriété a été enlevé, elle est trivialement vraie, et ce, même si le modèle est erroné. On doit au moins vérifier qu'il est possible de faire chaque type de transfert dont SINGLE/INCR/INCR4/INCR8/INCR16 et WRAP4/WRAP8/WRAP16, écriture/lecture, chaque taille de transfert permis et tester que l'esclave peut donner tous ses types de réponses.

La règle 4 de la liste obtenue (snpsAhbMasterChk_addr_cntrl_stable) dit que les signaux d'adresses et de contrôles doivent rester les mêmes durant une extension du temps de transfert par un esclave. Cela pose problème si le transfert est le dernier d'une rafale et que le maître fait un transfert de type IDLE suivi d'une nouvelle requête. La propriété interdit cela, mais la spécification ne l'interdit pas. Pour rendre la propriété valide, il faut que ce soit vrai sauf si le transfert est de type IDLE.

La règle 11a de la liste obtenue (snpsAhbMasterChk_cntrl_stable_addr_incrm) vérifie que pendant un transfert qui n'est pas le premier d'un transfert en rafale, les signaux de contrôle ne sont pas changés et que les signaux d'adresses changer selon le type de transfert. Dans la description de la propriété, il est dit que si le signal HTRANS est à SEQ et que le cycle précédent il était à NONSEQ, SEQ ou BUSY, l'adresse doit changer. Ce n'est pas vrai si dans le cycle précédent le signal HTRANS avait la valeur BUSY, car le transfert n'a pas été fait et que l'adresse a déjà changé le cycle précédent et ne doit pas changer à nouveau. Il y a un exemple de ce problème dans le graphique 3-6 de la spécification.

Dans la liste de règles, il est dit si la règle ne s'applique qu'à la version complète du bus. Par contre, cette indication n'est pas donnée pour certaines propriétés. C'est le cas pour la propriété 17 (SnpsAhbMasterChk_new_trans_nonseq_idle (again)), car le maître ne fait pas de requête pour le bus (spec d'AHB-Lite p.4). Aussi pour la propriété 23 (SnpsAhbMasterChk_complete_trans) qui vérifie qu'un transfert en rafale doit terminer à moins que le bus soit retiré au maître, car le maître ne fait pas

de requête pour le bus et ne peut donc pas le perdre (spec d'AHB-Lite p.4). C'est le cas pour la propriété 40 (SnpsAhbArbiterChk_valid_out) qui vérifie une propriété pour l'arbitre, car le bus AHB-Lite n'a pas d'arbitre (spec d'AHB-Lite p.4). Dernièrement, il y a la règle 52 (SnpsAhbArbiterChk_valid_out) qui est aussi une règle pour l'arbitre et donc n'a s'applique pas pour le bus AHB-Lite.

Finalement, il y a la règle 34 (SnpsAhbSlaveChk_resp_idl_busy), qui parle du signal HREADY. Le problème, comme discuté plus haut, est que ce signal à plusieurs significations : comme sortie de chaque esclave et comme le résultat du multiplexeur qui prend en entrée ces signaux.

5.7. Amélioration de l'outil

Il existe plusieurs améliorations possibles du logiciel utilisé CPN Tools. Une d'elles est de créer une fonction qui retourne à l'état précèdent de retour dans la simulation. Ensuite, si on avance à nouveau, on reprend le même chemin déjà fait. Cela permettrait d'utiliser davantage la simulation pour déboguer le système avant de faire les vérifications formelles. Par exemple, on pourrait, après une modification au modèle, exécuter la simulation rapidement pour détecter des inconsistances de types ou trouver des états terminaux. Le problème est que si on en trouve un, on ne sait pas par quel chemin on y est allé. Il est possible d'imprimer une trace de la simulation, mais elle n'est pas facile à utiliser pour débugger. Utiliser la fonctionnalité de retour simplifierait le débogage. Pour l'implémentation de cette fonctionnalité, il y a trois options possibles : permettre un retour infini de transition, permettre un nombre maximum d'étapes mémorisées ou n'en permettre aucun. Le choix qui semble le plus facile à implémenter et qui n'est pas trop coûteux en temps d'exécution est celle avec un nombre maximum d'étapes de retour. Il est possible de garder en mémoire juste la séquence d'éléments de liaison (binding element) pour retrouver les états précédents pour éviter de garder en mémoire tous les états précédents.

Ceci permet de faire tourner rapidement le simulateur pour trouver des interblocages et avoir le chemin qui le crée. C'est plus rapide que l'analyse du graphe d'états pour une vérification rapide de changement avant de faire l'analyse complète.

Chaque instance d'une sous-page a un numéro pour la différencier. Dans certains cas particuliers, les numéros peuvent changer. Utiliser des noms pour les instances simplifierait la compréhension

d'une trace d'exécution. Il serait possible de prendre le nom de la transition de substitution comme nom de l'instance. Il y aurait comme contrainte que chaque instance d'une transition de substitution d'une page doit être unique.

Idéalement il faudrait que toutes les analyses formelles puissent être faites avec le logiciel. Pour ce faire, le concepteur pourrait transformer son modèle vers des formats supportés dans les logiciels intégrés [14, 69] qui relient plusieurs outils de vérification. Malheureusement, transformer le modèle en réseau de Petri coloré vers d'autres modèles semble très difficile, voire impossible, car il est très riche à cause des fonctions SML.

Plusieurs fonctions existantes qui calculent des propriétés sur le graphe d'états ou le graphe de composantes fortement connexes recalculent la propriété chaque fois qu'elles sont appelées. Comme on appelle plusieurs fois certaines de ces fonctions sans régénérer les graphes, il serait intéressant que chaque fonction garde en mémoire la réponse du dernier calcul et si le graphe n'est pas modifié depuis la dernière fois qu'elle a été exécutée, qu'elle retourne la dernière réponse. Cela faciliterait l'utilisation de ces fonctions en nous évitant de mettre dans une cache la réponse pour avoir plus de vitesse.

Il y a trois fonctions qui donnent la liste des transitions avec les propriétés d'équité faiblement juste, fortement juste et impartiale. Il manque une fonction qui donne la liste des transitions qui n'ont aucune propriété d'équité. La première retourne une liste de transitions et la deuxième un ensemble multiple de transitions. Elle est par contre plus simple à créer.

```
fun ListNoFairnessTIs() = List.foldl (fn (x,l)=>(List.filter (fn n=>
(n=x)) l )) (TI.All) (ListJustTIs()) :TI.TransInst list;
fun ListNoFairnessTIs2() = TI.All -- ListJustTIs() :TI.TransInst ms;
```

Il existe une fonction qui donne le marquage d'une place pour un état donné : `fun Mark.<PageName>'<PlaceName> Inst -> (Node -> CS ms)`. Malheureusement, il n'y a pas de moyen pour avoir une liste de toutes ces fonctions pour vérifier automatiquement certaines propriétés et il n'y a pas d'autre fonction utilisable. Il existe déjà une fonction qui donne une liste de toutes les places (PI.All) et une fonction qui donne la liste de toutes les transitions (TI.All). Avoir une fonction équivalente pour le marquage des places ou qui prendrait un élément de la liste PI.ALL permettrait de vérifier automatiquement certaines propriétés, comme vérifier que toutes les places ont au maximum 1 jeton. Actuellement, il faut lister manuellement ces fonctions, cela risque d'entraîner

des erreurs, car si on ajoute des places, on doit les rajouter dans la liste.

Le chapitre suivant est la conclusion de ce travail et donne plusieurs pistes de développement futur.

6. Conclusion et discussion

6.1. Synthèse

Ce mémoire de maîtrise présente une méthodologie de haut niveau pour la vérification formelle de spécification de système électronique. On décrit comment cette méthodologie est utilisable avec les réseaux de Petri. Finalement, une étude de cas montre que la méthodologie est fonctionnelle avec les outils actuels. Cette vérification est d'une grande utilité pour les spécifications qui sont utilisées à grande échelle comme les standards, car les conséquences de requis manquant ou non précis peuvent être très coûteuses.

La méthodologie permet de vérifier la finitude, la complétude, la cohérence et des propriétés supplémentaires sur le modèle de la spécification. Le modèle est construit de manière incrémentale pour faciliter la détection de requis manquants et pour permettre au concepteur de voir quelle partie du modèle crée des explosions combinatoires. Cela donne des indices pour savoir quelle fonctionnalité le concepteur doit essayer d'abstraire, car l'explosion combinatoire est la problématique principale de notre méthodologie. Parmi les vérifications faites, certaines propriétés sont vérifiées avec le graphe de composantes fortement connexes pour éviter ou diminuer le problème d'explosion combinatoire. Si on utilise les réseaux de Petri, la structure du modèle peut être utilisée pour le même but.

Ensuite, l'implémentation de cette méthodologie avec les réseaux de Petri colorés est donnée. Ce modèle de calcul permet des propriétés de cohérence supplémentaire qui peuvent être vérifiées. Si la technique de modélisation utilisée est celle de ce mémoire, il y a encore d'autres propriétés de cohérence qui peuvent être vérifiées.

L'étude de cas a été faite sur le bus AHB-Lite avec l'outil CPN Tools. La complétude, la cohérence et le respect de propriétés dérivées de la spécification ont été vérifiés. Ce qui a été trouvé est que plusieurs sections ne sont pas claires. Entre autres, il y a un signal HREADY qui est utilisé, mais qui a plusieurs significations. La spécification ne contient pas de requis manquants. Enfin, les règles dérivées de la spécification qu'on a reçues avaient quelques problèmes mineurs.

6.2. Perspective

La plus grande difficulté reliée à la méthodologie basée sur les réseaux de Petri colorés n'est pas la méthodologie elle-même, mais plutôt le fait que les outils qui la supportent ne font pas toutes les vérifications proposées dans la méthodologie. Il y aurait du développement à faire dans cette direction. Il serait intéressant de changer de modèle de calcul pour des réseaux de Petri bien formés. Ce modèle de calcul est plus facile à transformer vers les autres modèles de calcul et, avec le standard PNML qui est en développement, il serait possible de réutiliser plusieurs outils développés indépendamment. Il y a présentement des logiciels intégrés qui vont dans cette direction, dont CPN-AMI [14] et le Model-Checking Kit [69]. Malheureusement, ils ont plusieurs problèmes, dont la difficulté d'utilisation et de modélisation. Il y a l'outil BRITNeY Suite [70] qui devrait inclure plus de vérification formelle sur les réseaux de Petri coloré dans une version ultérieure.

Transformer les réseaux de Petri colorés vers les réseaux bien formés n'est pas une approche intéressante, car les réseaux colorés permettent d'utiliser des fonctions SML et il faudrait convertir toute la puissance du langage fonctionnel SML vers les réseaux bien formés.

Une autre piste de travail possible est de trouver comment modéliser de manière générique en réseaux de Petri coloré des modèles asynchrones. En ce moment, c'est au concepteur de trouver comment le modéliser. Pour cela, il serait intéressant d'aller voir les travaux sur les schémas [71] en réseau de Petri coloré et voir si on peut les utiliser. On pourrait faire la même chose pour modéliser des systèmes avec plusieurs horloges.

Mais le plus intéressant à mon avis personnel est de lier cette méthodologie avec les méthodologies actuelles pour ne pas avoir à créer de modèle supplémentaire. Cela peut être de convertir le réseau de Petri dans des langages synthétisables ou l'inverse, traduire les langages synthétisables vers les réseaux de Petri pour faire leur vérification. Cela éviterait au concepteur d'apprendre un nouveau langage et pourrait être utilisé facilement avec les projets qui sont déjà existants. Par contre, cela perdrait la possibilité de détecter les requis manquants de la spécification, car cela travaille au niveau de l'implémentation et retirerait certaines propriétés qui sont nécessaires pour la détection des requis manquants.

Si on va dans la direction de la transformation de réseau de Petri dans un autre langage

synthétisable, le langage Bluespec System Verilog semble une bonne idée à première vue, car il est basé sur des règles et les réseaux de Petri peuvent être vus comme des règles. Comme il fait déjà plusieurs optimisations entre les règles, cela récupérerait leur travail d'optimisation, car cela risque d'être la partie la plus difficile de la transformation.

Explorer les techniques de re-écriture pour permettre au réseau de Petri d'avoir directement dans le langage des antiplaces, pour avoir des « vrais » paramètres, pour générer des propriétés qui doivent être valide, etc. Souvent cela n'ajoute rien à la puissance du modèle de calcul, mais c'est d'une grande utilité pour faciliter sont utilisation.

Références

[1] J. Martinez and M. Silva, "A simple and fast algorithm to obtain all invariants of a generalized Petri net," presented at 2nd European Workshop on Application and Theory of Petri Nets, Bad Honnef, West Germany, 1981.

[2] Intel, "FDIV Replacement Program Statistical Analysis of Floating Point Flaw," November 30, 1994.

[3] G. Le Lann, "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective," presented at International Conference and Workshop on Engineering of Computer-Based Systems, Monterey, CA, USA, 1997.

[4] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *Computer*, vol. 26, pp. 18-41, 1993.

[5] A. Sasongko, "Prototypage basé sur une platforme reconfigurable pour la vérification des systèmes monopuces," in *TIMA*, vol. Phd. Grenoble: Université Joseph Fournier, 2004, pp. 47.

[6] A. Bouhoula, E. Kounalis, and M. Rusinowitch, "SPIKE: An automatic theorem prover," presented at Proc. International Conference Logic Programming and Automated Reasoning, St. Petersburg (Russia), 1992.

[7] "HOL Proof Assistant," http://hol.sourceforge.net/.

[8] "Isabelle Theorem Proving Environment," http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[9] J.-P. Katoen, "Principles of Model Checking," 2004-2005.

[10] J. Mullins, "Principes et méthodes du model-checking," Polytechnique de Montréal, Montréal 2003.

[11] M. Metzger, F. Bastien, F. Rousseau, J. Vachon, and E. M. Aboulhamid, "Introspection Mechanisms for Semi-Formal Verification in a System-Level Design Environment," presented at Rapid System Prototyping (RSP2006), 2006.

[12] "CPN Tools," http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

[13] "GreatSPN," http://www.di.unito.it/~greatspn/index.html.

[14] "CPN-AMI," http://www-src.lip6.fr/logiciels/mars/CPNAMI/.

[15] W. Farn, "Formal verification of timed systems: a survey and perspective," *Proceedings of the IEEE*, vol. 92, pp. 1283-1305, 2004.

[16] "PSL/Sugar Consortium," http://www.pslsugar.org/.

[17] "System Verilog Assertion (SVA)," http://www.synopsys.com/products/simulation/assert_sverilog_wp.html.

[18] C. Liu, A. Kondratyev, Y. Watanabe, and A. Sangiovanni-Vincentelli, "A Structural Approach to QuasiStatic Schedulability Analysis of Communicating Concurrent Programs," presented at 5th ACM international conference on Embedded software (EMSOFT'05), Jersey City, New Jersay, USA, 2005.

[19] "OVL (Open Verification Library)," *http://www.accellera.org/activities/ovl*.

[20] J.-P. David, "Architecture synchronisée par les données pour système reconfigurable," in *Laboratoire de Microélectronique*, vol. Ph.D. Thesis: Université Catholique de Louvain, 2002.

[21] "A Classification of Petri Nets," http://www.informatik.uni-hamburg.de/TGI/PetriNets/classification/.

[22] K. Jensen, "Condensed State Spaces for Symmetrical Coloured Petri Nets," *Formal Methods in System Design 9*, vol. 9, pp. 7-40, 1996.

[23] "Petri Nets Tool Database http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html."

[24] K. Jensen, *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use: Basic Concepts*, vol. 1: Springer-Verlag, 1997.

[25] "Performance Analysis In CPN-Tool," http://wiki.daimi.au.dk/cpntools-help/performance_analysis.wiki.

[26] J. R. Agre and S. K. Tripathi, "Approximate solution to multichain queueing networks with state dependent service rates," pp. 45–55, 1985.

[27] M. Weber and E. Kindler, "The Petri Net Markup Language," in *Advances in Petri Nets*, vol. Petri Net Technology for Communication Based Systems, *LNCS*, 2002.

[28] J. Billington, S. Christensen, K. v. Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber, "The Petri Net Markup Language: Concepts, Technology, and Tools," presented at ICATPN Eindhoven, Netherlands, 2003.

[29] K. Jensen, "An Introduction to the Practical Use of Coloured Petri Nets," presented at Lectures on Petri Nets II: Applications, 1998.

[30] K. Jensen, "A Brief Introduction to Coloured Petri Nets," presented at Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97 Workshop), Enschede, The Netherlands, 1997.

[31] L. M. Kristensen, S. Christensen, and K. Jensen, "The Practitioner's Guide to Coloured Petri Nets," *International Journal on Software Tools for Technology Transfer*, vol. 2, pp. 98-132, 1998.

[32] K. Jensen, "An Introduction to the Theoretical Aspects of Coloured Petri Nets," presented at A Decade of Concurrency, 1994.

[33] K. Jensen, *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use: Practical Use*, vol. 3: Springer-Verlag, 1997.

[34] K. Jensen, *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use: Analysis Methods*, vol. 2: Springer-Verlag, 1997.

[35] H. J. Genrich and R. M. Shapiro, "Formal Verification of an Arbiter Cascade," presented at 13th International Petri Net Conference, Sheffield, 1992.

[36] F. Burns, A. Koelmans, and A. Yakovlev, "Analysing Superscalar Processor Architectures with Coloured Petri Nets," *International Journal on Software Tools for Technology Transfer*, vol. 2, pp. 182-191, 1998.

[37] R. M. Shapiro, "Validation of a VLSI Chip Using Hierarchical Coloured Petri Nets," *Journal of Microelectronics and Reliability, Special Issue on Petri Nets*, pp. 667-687, 1991.

[38] "SystemC," http://www.systemc.org/.

[39] "ESys.Net," http://www.esys-net.org/.

[40] M. Metzger, "Mécanismes d'introspection pour la vérification semi-formelle de modèles au niveau système," in *DIRO*, vol. Master. Montréal: Université de Montréal, 2006, pp. 154.

[41] "Cynthesizer," http://www.forteds.com/products/cynthesizer.asp.

[42] G. Dec, "Automatic VHDL Generation from XML Description of Fuzzy Petri Net," *Petri Net Newsletter*, pp. 88 p., 2005.

[43] N. Gorse, P. Belanger, E. M. Aboulhamid, and Y. Savaria, "A High-Level Requirements Engineering Methodology for Electronic System-Level Design," *Special issue of the International Journal on Computers in Electrical Engineering*, 2005.

[44] G. Rob, P. Doron, Y. V. Moshe, and W. Pierre, *Simple on-the-fly automatic verification of linear temporal logic*: Chapman & Hall, 1996.

[45] "PROD," http://www.tcs.hut.fi/Software/prod/.

[46] N. Gorse, "Méthodes formelles de haut niveau pour la conception de systèmes électroniques fiables.," in *DIRO*, vol. Phd. Montréal: Université de Montréal, 2006.

[47] L. Åqvist, "Introduction to Deontic Logic and the Theory of Normative Systems," *Bibliopolis*, 1983.

[48] M. Barbuceanu, T. Gray, and S. Mankowski, "How To Make Your Agents Fulfil Their Obligations," presented at PAAM-98, London, UK, 1998.

[49] D. Lehmann, A. Pnueli, and J. Stavi, "Impartiality, Justice and Fairness: the Ethics of Concurrent Termination," presented at ICALP 1981, 1981.

[50] K. Varpaaniemi, "On Stubborn Sets in the Verification of Linear Time Temporal Properties," *Formal Methods in System Design 9*, vol. 26, pp. 45--67, 2005.

[51] "Specification Patterns," http://patterns.projects.cis.ksu.edu/.

[52] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in Property Specification for Finite-State Verification," presented at Proceedings of the 21st International Conference on Software Engineering, May 1999.

[53] "Maria," http://www.tcs.hut.fi/Software/maria/.

[54] S. Christensen and K. H. Mortensen, "Design/CPN ASK-CTL Manual Version 0.9," University of Aarhus 1996.

[55] A. Cheng, S. Christensen, and K. Mortensen, "Model Checking Coloured Petri Nets Exploiting Strongly Connected Components," University of Aarhus, Computer Science Department March 1997.

[56] A. A. Tovchigrechko, "Analysis of Bounded Petri Nets Using Interval Decision Diagram," *Petri Net Newsletter*, vol. 70, pp. 21-30, 2006.

[57] A. Limited, "AHB-Lite Overview," Copyright: ARM Limited. All rights reserved 2001.

[58] A. Limited, "AMBA Specification (Rev 2.0)," Copyright: ARM Limited. All rights reserved 1999.

[59] "Examples of Industrial Use of CPN for Hardware," http://www.daimi.au.dk/CPnets/intro/3.html.

[60] "Examples of Industrial Use of CPN for Software," http://www.daimi.au.dk/CPnets/intro/2.html.

[61] J. Schmaltz and D. Borrione, "Validation of a Parameterized Bus Architecture Model," presented at Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003), in Boulder, Colorado, USA, 2003.

[62] "ACL2," http://www.cs.utexas.edu/users/moore/acl2/.

[63] H. Amjad, "Verification of AMBA using a combination of model-checking and theorem proving," presented at Proceedings of the 5th International Workshop on Automated Verification of Critical Systems, 2005.

[64] H. Amjad, "Model checking the AMBA protocol in HOL," september 2004.

[65] M. C. Newey, "A Z specification of the AMBA high-performance bus DRAFT?," Department of Computer Science, The Australian National University May 31 2004.

[66] A. Roychoudhury, T. Mitra, and S. R. Karri, "Using formal techniques to debug the AMBA system-on-chip bus protocol," presented at Design, Automation and Test in Europe Conference and Exhibition, 2003.

[67] K. Jensen, S. Christensen, and L. M. Kristensen, "CPN Tools Occurrence Graph Manual," University of Aarhus 2002.

[68] M. Metzger, F. Bastien, F. Rousseau, J. Vachon, and E. M. Aboulhamid, "Semi-Formal Verification Tool Implementation using Introspection Mechanisms," presented at Forum on specification & Design Languages(FDL2006), TU Darmstadt, Germany, 2006.

[69] "Model-Checking Kit," http://www.informatik.uni-stuttgart.de/fmi/szs/tools/mckit/.

[70] "BRITNeY Suite," http://wiki.daimi.au.dk/britney/britney.wiki.

[71] N. Mulyar and W. M. P. v. d. Aalst, "Towards a Pattern Language for Colored Petri Nets," presented at Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'05), University of Aarhus, Denmark, 2005.

Annexe A : Glossaire

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced System Bus
ASB	Advanced Peripheral Bus
BDD	Binary Decision Diagram (Fr) Diagramme de décision binaire
Binding(An)	Liaison
BMC	Bounded Model Checking
Burst(An)	(Fr) Transfert en rafale
CPN	Colored Petri Net, Réseau de Petri coloré (Fr)
CSP	Communicating Sequential Processes
CTL	Computational Tree Logic (Fr) Logique computationnelle arborescente
Deadlock(An)	Inter-blocage
Don't care(An)	Sans importance, qui n'affecte en rien le comportement de la spécification
EDA(An)	Electronic Design Automation (Fr) Conception électrique assistée par ordinateur
Équité(Fr) (Propriété)	(An) Justice (Property)
Fairness(An)	Équitable
FPGA(An)	Field Programmable Gate Array
FSM(An)	Finite State Machine (Fr) Machine à états
Fuzzy(An)	(Fr) Flou
HOL(An)	Higher Order Logic (Fr) Logique d'ordre supérieur
LTL(An)	Linear temporal logic

	(Fr) Logique linéaire temporel
Model checking(An, Fr)	Méthode de vérification formelle basée sur le graphe d'états
OVA(An)	Open Vera Assertion
OVL(An)	Open Verification Library
OPI	Ensemble de propriétés qui inclut l'Obligation, la Permission et l'Interdiction
PEPS	Premier entré, premier sorti (An) First in, first out (FIFO)
PN(An)	Petri Net (Fr) Réseau de Petri (RdP)
Proof checkers(An)	Vérification de théorème
PSL	Property Specification Language
Rafale	Groupe de données transmises en une seule fois. (En) Burst
RdP	Réseau de Petri (An) Petri Net (PN)
Reachability graph(An)	(Fr) Graphe d'états [atteignable], graphe des marquages accessibles
Requis(An)	Pré requis, exigence
RG(An)	Reachability Graph (Fr) Graphe d'états [atteignables]
SAT(An)	Satisfiability
SCC(An)	Strongly Connected Component (Fr) Composante fortement connectée Un SCC est un sous-graphe du graphe d'états où il existe un chemin entre tous les noeuds A strongly connected component is a subgraph in which there exists a directed path from any node to any node
SMV	Symbolic Model Verifier (Tool) (Fr) Outils de vérification symbolique de modèle
SVA	System Verilog Assertion

SWN	Stochastic Well-formed Net (Fr) Réseau de Petri bien formé stochastique
Theorem Prover(An)	(Fr) Prouveur de théorème
Verification on the fly(An)	(Fr) Vérification à la volée Vérification de propriétés pendant la génération du graphe d'états à l'opposé de vérifier les propriétés une fois le graphe d'états généré.
WN	Well-formed Net (Fr) Réseau de Petri bien formé