Construction de Modèles Réduits et Vérification Symbolique de Circuits Industriels décrits au Niveau RTL

E. Dumitrescu

UNIVERSITÉ JOSEPH FOURIER - GRENOBLE 1

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Discipline : MICROÉLECTRONIQUE

École Doctorale Electronique, Electrotechnique, Automatique, Télécommunications, Signal

présentée et soutenue publiquement

par

Emil DUMITRESCU

le 7 Octobre 2003

Construction de modèles réduits et vérification symbolique de circuits industriels décrits au niveau RTL

Directeur de thèse : Dominique Borrione

JURY

Rapporteurs	Mr Hans EVEKING,	Prof. Université de Darmstadt
	Mr Charles ANDRÉ,	Prof. Université de Nice
Examinateurs	Mr Nicolas HALBWACHS,	Directeur de Recherche CNRS
	Mr Christian BERTHET,	ST-Microelectronics
Directeur de thèse	Mme Dominique BORRIONE,	Prof. Université Joseph Fourier

Thèse préparée àu Laboratoire **Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs (TIMA)**

Remerciements

Je souhaite m'adresser en premier lieu aux membres du jury de thèse. Je remercie donc Messieurs Hans EVEKING et Charles ANDRE pour avoir accepté le travail de rédaction des rapports de pré-soutenance. Merci pour la relecture approfondie du manuscrit, ainsi que pour leurs remarques constructives et leurs encouragements.

Je remercie Messieurs Nicolas HALBWACHS et Christian BERTHET pour m'avoir fait l'honneur de faire partie de ce jury.

Je remercie enfin Mme Dominique BORRIONE. Toute ma reconnaissance pour son accueil exceptionnel au sein de l'équipe VDS, son soutien et sa patience, et pour l'ensemble des opportunités qu'elle m'a offertes pendant toutes ces années.

Je souhaite maintenant adresser toute ma sympathie à mes collègues du laboratoire TIMA, en particulier aux membres (encore présents, ou absents à ce jour) de l'équipe VDS. C'est à eux que je dois des moments inoubliables, passés dans un cadre aussi bien amical, bienveillant et drôle, que passionné, encourageant et enrichissant pour mon travail. Un grand merci donc à Menouer BOUBEKEUR, Diana TOMA, Julien SCHMALTZ, Ghiath AL-SAMMANE, Eric GASCARD, Philippe GEORGELIN et Cyrille CHAVET.

Je remercie Pierre OSTIER pour son expérience de chercheur qu'il n'a pas hésité à partager, ainsi que pour son assistance en tant qu'ingénieur système (notamment lors d'incidents matériels, produits les week-ends ou tard le soir, à l'époque où je rédigeais ce manuscrit). Mais je lui dois surtout une sacrée dose d'encouragements, de conseils (de toute sorte...), de bonne humeur, de soutien et d'amitié. Je garde un souvenir très agréable de cette longue période durant laquelle nous avons partagé le même bureau...

Un grand merci (en forme de saxophone ténor) à Claude LE FAOU. Merci pour ses conseils, son aide bienveillante, sa disponibilité et sa patience (envers nous tous d'ailleurs). Mais Claude a été pour nous plus que tout cela. C'est grâce à lui qu'il existe un moment privilégié dans chaque journée, nous permettant de nous retrouver tous autour d'un excellent café (dont il a conservé le secret des fournisseurs). J'ajoute mon grief vis-à-vis de son départ à la retraite...

Mes remerciements à Bénédicte FLUXA, et toute ma gratitude pour l'aide qu'elle m'a apportée depuis son arrivée au sein de l'équipe VDS.

Un grand merci à Eduard (Ed) CERNY et à Christian BERTHET, pour leur soutien et leurs encouragements durant mes différents "séjours" à ST-Microelectronics. J'ajoute que le temps passé là-bas a été extrêmement profitable pour l'avancement de mon travail de thèse. J'adresse également mes remerciements et mes amitiés à Sébastien FRANCOIS, Stéphane BREFORT, Laurence ALBACETE, Philippe DEBAUD, Danielle JACOB, Bruno DENIS, Xavier CHBANI, Fabrice PICHAT, Karim DAHMANE et Nathalie DESCHAMPS.

Enfin, je remercie les membres de l'équipe CAD du département EECS de l'Université de Californie à Berkeley pour leur accueil chaleureux durant mon séjour dans leur équipe, en tant qu'invité. Merci à Robert BRAYTON, Flora OVIEDO, Adrian ISLES, Sriram KRISHNAN, Sriram RAJAMANI, Sunil KHATRI et Luciano LAVAGNO.

Table des matières

Introduction

Dans le contexte du développement des circuits numériques, le niveau de description communément utilisé pour la conception et accepté à la fois par les outils de simulation, de synthèse et de vérification est le niveau "transfert de registres" (RTL). Par conséquent, la vérification de descriptions RTL est une étape à la fois prédominante et incontournable. La complexité croissante des circuits réalisés requiert des méthodes complémentaires pour appuyer la simulation, dans le but de renforcer la fiabilité de la conception du circuit. La *vérification de modèles* est une technique de preuve formelle qui a émergé au cours des dernières années, et qui a pu être appliquée avec succès à des projets industriels. Par conséquent, cette technique a été mise en œuvre au sein de nombreux outils commerciaux ou du domaine public. A l'heure actuelle, la vérification formelle de descriptions RTL bénéficie d'un cadre automatisé et relativement facile d'utilisation. Pourtant, les limites imposées par sa complexité exponentielle rendent son utilisation problématique. Etant donné un circuit de taille importante et une propriété à vérifier, il est impossible, pour un utilisateur non expert, de prévoir si l'évaluation de la propriété pourra se faire dans les limites des ressources (temps et mémoire) disponibles. De ce fait, l'application de la vérification de modèles est aujourd'hui limitée à des circuits de taille très réduite[1], ne contenant pas plus de quelques centaines de registres.

L'usage de spécifications plus abstraites et exécutables permettrait de décrire les caractéristiques essentielles du circuit que l'on souhaite élaborer, tout en limitant la complexité de la représentation. C'est dans ce contexte que la vérification de modèles serait plus efficace. Une étape de génération telle que la synthèse "de haut niveau" permettrait d'obtenir une description RTL synthétisable à partir d'une telle spécification. Cette étape devrait préserver la correction de toutes les propriétés déjà vérifiées sur la spécification. A ce jour, la synthèse "de haut niveau", lorsqu'elle est applicable, ne garantit pas que la description RTL produite vérifie de la même façon les propriétés évaluées sur la spécification initiale. Il est donc nécessaire de procéder aussi à la vérification de la description RTL résultante. Ainsi, la mise en œuvre de méthodes efficaces de vérification pour le niveau RTL demeure un sujet de grande actualité.

Au niveau RTL, les méthodes fondamentales de vérification de modèles ne permettent pas d'aboutir à des résultats dans des délais raisonnables. C'est pourquoi l'intervention manuelle de l'ingénieur de vérification est essentielle, pour guider la vérification afin d'éviter l'explosion combinatoire : il doit apporter sa connaissance sur le circuit vérifié, sur les

[1] à l'échelle industrielle...

formules temporelles à écrire, et suggérer une stratégie adéquate de réduction de complexité de la description manipulée.

Une partie du travail de cette thèse a été menée dans un contexte industriel. A travers cette expérience, qui a permis de comparer différents outils de vérification de modèles, commerciaux et du domaine public, l'étude a été axée sur le développement de méthodes permettant d'utiliser cette technique de manière plus efficace. La principale contribution de ce travail porte sur la réduction, manuelle ou non, du modèle vérifié. La *vérification compositionnelle* est un concept répandu, permettant de décomposer le circuit vérifié suivant un critère structurel, suggéré par la hiérarchie de modules du circuit vérifié. Nous avons appliqué cette technique dans un cadre général, et nous l'avons adaptée à une situation réelle très importante : lorsque le module vérifié est très complexe, tout en étant un nœud terminal de la hiérarchie de modules, la décomposition structurelle devient inapplicable quoique nécessaire. En nous appuyant sur une connaissance a priori du circuit vérifié ainsi que sur sa description, nous présentons et illustrons une approche, intitulée *méthode ascendante*, permettant d'identifier ses fonctionnalités sous la forme de blocs fonctionnels, et de les vérifier ensuite selon l'approche compositionnelle classique.

La décomposition de la vérification peut également se faire dans le domaine comportemental. Encore une fois, en s'appuyant sur une connaissance a priori du circuit, on identifie différents *scénarios d'exécution*, spécifiés uniquement à l'état initial (mise sous tension) du circuit vérifié. Cette approche permet d'engendrer autant de modèles réduits que de scénarios d'exécution, pouvant être vérifiés séparément.

Notre principale contribution a été l'extension de ce raisonnement comportemental. Le comportement de certains circuits peut être *décomposé de façon séquentielle*. Dans une approche classique de fonctionnement, l'environnement doit produire une séquence de valeurs sur certaines entrées du circuit afin de mettre ce dernier dans un mode de fonctionnement "normal". Cette opération peut être synonyme de remise à zéro, ou encore de programmation du circuit afin de lui transmettre un nouveau paramétrage. Une fois la remise à zéro effectuée, ou le paramétrage terminé, le circuit entre dans le mode de fonctionnement "normal".

Cette méthode de décomposition est appelée *partition fonctionnelle*. Sa mise en place s'appuie sur la technique de *simulation symbolique*. Elle a été appliquée sur un circuit de taille importante et a permis d'obtenir des résultats très prometteurs en matière de réduction.

A partir de circuits décrits dans le langage VHDL, l'expérimentation des techniques de preuve offertes par les différentes outils de vérification de modèles s'est appuyé sur un outil d'extraction de machines d'états finis qu'il a été nécessaire de mettre en œuvre.

Ce document est divisé en deux parties principales. La première partie, intitulée *Représentation et Algorithmes*, présente le niveau propositionnel booléen, utilisé pour la représentation des circuits numériques et pour parcourir leur espace d'états. Ensuite, on présente les techniques de spécification formelle rencontrées dans l'industrie et les algorithmes de preuve associés. On discute également des techniques automatiques de réduction du modèle pour la preuve. Ces techniques fondamentales sont ensuite comparées du point de vue de leur efficacité, de leur pouvoir d'expression et de l'aisance de leur manipulation.

La deuxième partie de ce document, intitulée *Stratégies de vérification*, est consacrée

aux stratégies d'application de la vérification de modèles sur des circuits industriels. A ce niveau, il s'agit de manipulations à caractère plus ou moins systématique qui requièrent l'intervention de l'utilisateur (en la personne de l'ingénieur de vérification). Après la présentation des techniques compositionnelles directes et des raisonnements associés, on introduit le principe de la méthode ascendante, méthode illustrée en détail au chapitre 4 de ce document. On discute ensuite de l'application des techniques comportementales, et on formalise la partition fonctionnelle, l'étape de réduction basée sur la simulation symbolique.

Enfin, le dernier chapitre de ce document présente la mise en œuvre des stratégies introduites et des outils développés. Tout d'abord, on introduit le sous-ensemble du langage VHDL sur lequel nous avons travaillé, et on expose les concepts de base de l'extraction de machines d'états à partir des descriptions conformes à ce sous-ensemble. L'étape d'extraction est discutée du point de vue de la sémantique de simulation du langage VHDL.

La suite est consacrée au traitement d'un exemple industriel de taille importante : un contrôleur de cache d'instructions conçu par la société ST-Microelectronics et que nous avons pu étudier. Ce module a été vérifié au sein d'une équipe de développement, dans le cadre d'un projet industriel de vérification. Puis, dans un but purement expérimental, il a été vérifié selon plusieurs approches et à l'aide de différents outils de vérification. Cette démarche est exposée sur une partie des propriétés de ce module, et ses résultats sont ensuite discutés.

Première partie

Représentation et Algorithmes

Chapitre 1

Représentation symbolique des machines d'états finis

1.1 Introduction

Comme dans toute pratique scientifique ou technique, le *modèle* constitue le pilier central dans le processus de conception d'un circuit numérique. C'est le liant qui confère à la fois forme, intelligibilité et communicabilité aux longues étapes de raisonnement qui précèdent sa réalisation. L'approche de modélisation raisonnablement rodée aujourd'hui est axée sur le modèle conceptuel d'un système à états, inséré dans un environnement adapté. Dans un tel système, un seul état, appelé état courant, est actif à un instant donné. Les transitions(changements d'état) sont déclenchées lorsqu'un événement attendu se produit dans l'environnement. En particulier, cet événement peut signifier "passage du temps", mis en oeuvre par un mécanisme appelé *horloge*. Ainsi, toute étape dans le flot de conception est centrée autour d'un tel modèle, dont la "qualité" formelle varie, en s'améliorant au fur et à mesure que l'on s'approche du résultat final. La spécification produit un modèle semi-formel de système à transitions, souvent à la fois abstrait et incomplet. Ce modèle sert de base au codage. Le résultat est une description HDL [1] ayant une sémantique formelle pour la simulation, mais sur laquelle le concepteur préférera garder une vue proche de son modèle spécifié. Un ensemble de fonctions logiques sont extraites de cette description, qui constituent un modèle formel pour la vérification et/ou la synthèse.

Tout au long de ce processus constitué de raffinements successifs, se pose le problème de la *validation*. Une fois la spécification effectuée, elle est aussitôt mise en doute : "ma spécification correspond-elle vraiment à ce que je souhaite réaliser ?". Puis, une fois le circuit manuellement codé "est-ce que ma description est restée fidèle à sa spécification ?". Des techniques de raisonnement et de preuve peuvent donner une réponse à ces questions, à condition que le modèle qui leur est fourni et qui sert de base à ce raisonnement soit décrit de manière formelle.

Le modèle formel le plus adapté à la conception de circuits numériques ainsi qu'à la preuve par vérification de modèles est celui de Machine d'Etats Finis (que l'on abrégera FSM) de Mealy. Ce chapitre est consacré au rappel du modèle hiérarchique de Mealy,

[1] abrégé en anglais : *Hardware Description Language*

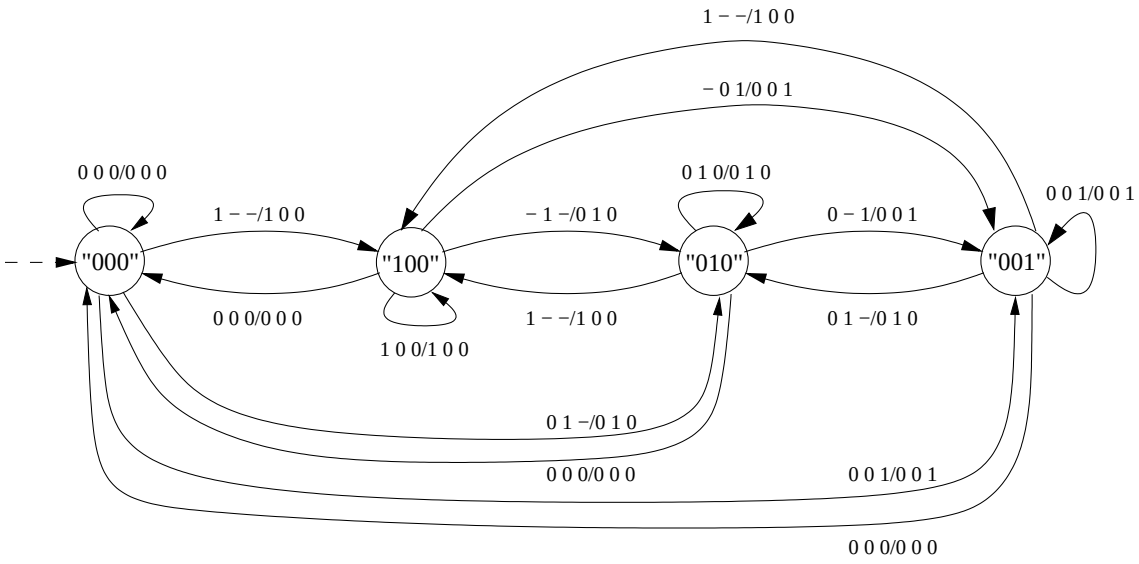

FIG. 1.1 – Arbitre à trois entrées

ainsi que de sa représentation symbolique dans la logique propositionnelle. Sont présentées ensuite les étapes de calcul symbolique qui constituent la base des algorithmes de preuve formelle. Il est également fait rappel des méthodes de représentation qui ont permis d'étendre le champ d'application de la vérification de modèle vers la preuve de circuits industriels.

Dans ce chapitre, la présentation des différents concepts est illustrée à l'aide de l'exemple ci-dessous : le modèle d'un arbitre, mettant en oeuvre un mécanisme de priorités, avec l'intention de servir l'ensemble de ses clients d'une manière équitable.

Exemple : un arbitre à trois entrées. La figure 1.1 présente le graphe de transition d'un arbitre qui gère les requêtes de trois clients différents. Ces clients sont classés par ordre de priorité du premier (le plus prioritaire) au troisième (le moins prioritaire). Cet exemple est inspiré de [62]. A chaque instant, cet arbitre lit et renvoie un symbole composé de trois chiffres binaires. Le symbole d'entrée représente l'état (demandeur ou non) de chacun de ses clients. Ainsi, le symbole "000" est lu si aucun client n'est demandeur d'accès. Si les clients 2 et 3 sont demandeurs en même temps, l'arbitre lira le symbole "011". Selon à la fois la valeur du symbole d'entrée et les décisions prises au cours du passé, cet arbitre donne une réponse d'acceptation : le symbole "000" est renvoyé si aucun client n'a reçu de permission d'accès ; si l'autorisation est accordée au client 1, l'arbitre renvoie le symbole "100". Cet arbitre met en œuvre l'exclusion mutuelle d'accès pour ses trois clients : tous les symboles sortants contiennent au maximum un seul chiffre "1". Le concepteur a également souhaité que le traitement des requêtes entrantes se fasse de manière équitable, condition qui n'est remplie que partiellement. Les étiquettes associées aux états de cette machine montrent le symbole sortant précédemment envoyé. Ainsi, l'étiqueté d'état "001" signifie que le client 3 a reçu précédemment la permission d'accès et sera de ce fait moins prioritaire que les clients 2 et 1. A la mise sous tension, l'arbitre se trouve dans l'état "000". Pour chacun des états,

une et une seule transition associée est toujours active. Chaque transition est étiquetée avec le symbole lu en entrée qui provoque son activation, suivi par le symbole de réponse renvoyé vers l'environnement. L'emploi du caractère '-' (signifiant valeur indifférente) permet de manipuler des ensembles de symboles d'entrée et de regrouper plusieurs transitions ayant le même symbole sortant. Par exemple, l'étiquette "- 0 1/0 0 1" représente les transitions "0 0 1/0 0 1" et "1 0 1/0 0 1".

1.2 Le modèle de machine d'états finis

1.2.1 Notions préliminaires

Les notations adoptées sont inspirées de [38]. Les fondements théoriques des machines d'états finis sont présentés dans [42, 56]. Cette section rappelle les définitions nécessaires pour la suite de ce document.

Définition 1.1 (Machine d'états finis) *Une machine d'états finis déterministe est un n-uplet*

$$\mathcal{M} = \langle \mathcal{X}, \mathcal{O}, \mathcal{S}, \sigma_0, \mathcal{F}_t, \mathcal{F}_o \rangle$$

où $\mathcal{X}$ est un ensemble fini de symboles d'entrée, $\mathcal{O}$ est un ensemble fini de symboles de sortie, $\mathcal{S}$ est un ensemble fini d'états, et $\sigma_0 \in \mathcal{S}$ est appelé l'état initial de $\mathcal{M}$. $\mathcal{F}_t : \mathcal{X} \times \mathcal{S} \to \mathcal{S}$ représente la fonction de transition d'état *de $\mathcal{M}$ et $\mathcal{F}_o : \mathcal{X} \times \mathcal{S} \to \mathcal{O}$ représente sa* fonction de sortie.

Tout le long de ce document, sauf avis contraire, nous nous restreignons aux machines d'états finis $\mathcal{M}$:

- *déterministes* : $\mathcal{F}_t$ doit être une fonction : elle associe à un couple de valeurs donné $\langle \xi, \sigma \rangle \in \mathcal{X} \times \mathcal{S}$ une unique valeur de $\mathcal{S}$;
- *complètes* : à la fois $\mathcal{F}_t$ et $\mathcal{F}_o$ sont définies sur la totalité de l'ensemble $\mathcal{X} \times \mathcal{S}$.

Si $\mathcal{F}_t$ est une relation telle que $\exists \xi \in \mathcal{X}, \exists \sigma, \sigma'_1, \sigma'_2, \sigma'_1 \neq \sigma'_2 : \sigma'_1 = \mathcal{F}_t(\xi, \sigma)$ et $\sigma'_2 = \mathcal{F}_t(\xi, \sigma)$, la machine est *non-déterministe*.

Habituellement, une machine d'états $\mathcal{M}$ est associée à une échelle discrète de temps. Ainsi, on associe l'instant "0" avec l'état initial σ_0 et on dit qu'à l'instant "0", $\mathcal{M}$ se trouve dans l'état σ_0. A tout instant T, $\mathcal{M}$ peut se trouver dans un état $\sigma_T \in \mathcal{S}$.

Définition 1.2 (Configuration courante) *A tout instant T, une machine $\mathcal{M}$ se trouve dans une configuration $C_T = \langle \xi_T, \sigma_T, o_T \rangle$ telle que $\xi_T \in \mathcal{X}$, $\sigma_T \in \mathcal{S}$, $o_T \in \mathcal{O}$ et $\mathcal{F}_o(\xi_T, \sigma_T) = o_T$.*

Définition 1.3 (Etat successeur) *Soient $\sigma, \sigma' \in \mathcal{S}$ deux états d'une machine $\mathcal{M}$. On dit que σ' est un successeur de σ, $\sigma \to \sigma'$ si et seulement si $\exists \xi \in \mathcal{X} : \sigma' = \mathcal{F}_t(\xi, \sigma)$.*

Définition 1.4 (Exécution) *Une exécution d'une machine d'états $\mathcal{M}$ est une séquence finie ou non de configurations $(C_0, C_1, C_2, ...)$ telle que $\forall k \in \mathbb{N}$ où $C_k = \langle \xi_k, \sigma_k, o_k \rangle$, $\sigma_{k+1} = \mathcal{F}_t(\xi_k, \sigma_k)$.*

Définition 1.5 (Configuration atteignable) *Une configuration $C = \langle \xi, \sigma, o \rangle$ est atteignable dans une machine $\mathcal{M}$ s'il existe une exécution $(C_0, ...C)$ où $C_0 = \langle \xi_0, \sigma_0, o_0 \rangle$ est une configuration dans laquelle $\mathcal{M}$ est à son état initial σ_0.*

Définition 1.6 (Machines d'états compatibles) *Soient*
$\mathcal{M}_1 = \langle \mathcal{X}_1, \mathcal{O}_1, \mathcal{S}_1, \sigma_{01}, \mathcal{F}_{t1}, \mathcal{F}_{o1} \rangle$ et $\mathcal{M}_2 = \langle \mathcal{X}_2, \mathcal{O}_2, \mathcal{S}_2, \sigma_{02}, \mathcal{F}_{t2}, \mathcal{F}_{o2} \rangle$ deux machines d'états finis éventuellement non-déterministes. On dit que $\mathcal{M}_1$ et $\mathcal{M}_2$ sont compatibles si et seulement si elles ont la même interface : $\mathcal{X}_1 = \mathcal{X}_2$ et $\mathcal{O}_1 = \mathcal{O}_2$.

Définition 1.7 (Etats équivalents) *Deux états $\sigma_1 \in \mathcal{M}_1$ et $\sigma_2 \in \mathcal{M}_2$ sont dits* équivalents, *$\sigma_1 \equiv \sigma_2$, si et seulement si $\mathcal{M}_1$ et $\mathcal{M}_2$ sont compatibles et $\forall \xi \in \mathcal{X}_1, \mathcal{F}_{o1}(\xi, \sigma_1) = \mathcal{F}_{o2}(\xi, \sigma_2)$.*

Définition 1.8 (Equivalence séquentielle) *Deux machines d'états finis $\mathcal{M}_1$ et $\mathcal{M}_2$ sont dites* séquentiellement équivalentes *$\mathcal{M}_1 \equiv \mathcal{M}_2$ si et seulement si les conditions suivantes sont satisfaites :*

- *$\mathcal{M}_1$ et $\mathcal{M}_2$ sont compatibles ;*
- *leurs états initiaux respectifs sont équivalents : $\sigma_{01} \equiv \sigma_{02}$;*
- *$\forall \sigma_1 \in \mathcal{S}_1$ et $\sigma_2 \in \mathcal{S}_2$, si $\sigma_1 \equiv \sigma_2$ alors $\forall \xi \in \mathcal{X}_1, \mathcal{F}_{t1}(\xi, \sigma_1) \equiv \mathcal{F}_{t2}(\xi, \sigma_2)$.*

Comportements non-déterministes. Du point de vue conceptuel, nous ne ferons pas de distinction entre une machine déterministe et une machine non-déterministe. Il est en effet possible de modéliser un comportement non-déterministe à l'aide d'une machine d'états déterministe, grâce à l'emploi d'un ensemble auxiliaire $\mathcal{ND} \neq \emptyset$ de symboles d'entrée, appelés des *pseudo-entrées*. Ainsi, une machine d'états finis non-déterministe $\mathcal{M}_{nd} = \langle \mathcal{X}, \mathcal{O}, \mathcal{S}, \sigma_0, \mathcal{G}_t, \mathcal{F}_o, \rangle$ où $\mathcal{G}_t$ est une relation de transition, peut être représentée par une machine déterministe $\mathcal{M}_d = \langle \mathcal{X} \times \mathcal{ND}, \mathcal{O}, \mathcal{S}, \sigma_0, \mathcal{F}_t, \mathcal{F}_o, \rangle$. La fonction de transition de la machine déterministe $\mathcal{M}_d$ sera définie par $\mathcal{F}_t : \mathcal{X} \times \mathcal{ND} \times \mathcal{S} \to \mathcal{S}$. $\mathcal{G}_t$ est la projection[2] de $\mathcal{F}_t$ parallèlement à $\mathcal{ND}$.

Ce procédé de modélisation du non-déterminisme introduit une contrainte d'unicité qui s'applique aux ensembles des pseudo-entrées : lorsque deux machines d'états sont comparées, leurs ensembles de pseudo-entrées respectifs doivent être disjoints. Les critères de compatibilité et d'équivalence séquentielle s'étendent facilement au cas non-déterministe.

Une relation typique entre deux machines d'états finis contenant du non-déterminisme est la simulation :

Définition 1.9 (Relation de simulation entre états) *Soient $\mathcal{M}_1$ et $\mathcal{M}_2$ deux machines d'états finis compatibles, éventuellement non-déterministes. L'état $\sigma_2 \in \mathcal{S}_2$* simule *l'état $\sigma_1 \in \mathcal{S}_1$, $\sigma_2 \preceq_s \sigma_1$, si et seulement si :*

- *$\sigma_1 \equiv \sigma_2$;*
- *$\forall \sigma'_1 \in \mathcal{S}_1, \forall \langle \xi, nd_1 \rangle \in \mathcal{X}_1^{nd}$ tel que $\sigma'_1 = \mathcal{F}_{t1}(\langle \xi, nd_1 \rangle, \sigma_1)$:*
  $\exists \sigma'_2 \in \mathcal{S}_2, \exists nd_2$ tels que $\langle \xi, nd_2 \rangle \in \mathcal{X}_2^{nd}$ et $\sigma'_2 = \mathcal{F}_{t2}(\langle \xi, nd_2 \rangle, \sigma_2)$ et $\sigma'_2 \preceq_s \sigma'_1$

[2] $\forall \xi \in \mathcal{X}, \forall \sigma \in \mathcal{S}, \sigma' \in \mathcal{G}_t(\xi, \sigma) \Leftrightarrow \exists nd \in \mathcal{ND}, \sigma' = \mathcal{F}_t(\xi, nd, \sigma)$

Définition 1.10 (Relation de simulation entre machines d'états finis) *Soient $\mathcal{M}_1$ et $\mathcal{M}_2$ deux machines d'états finis éventuellement non-déterministes. La machine $\mathcal{M}_2$ simule $\mathcal{M}_1$, $\mathcal{M}_2 \preceq_s \mathcal{M}_1$ si et seulement si $\sigma_{02} \preceq_s \sigma_{01}$*

Remarque. Lorsque deux machines d'états $\mathcal{M}_1$ et $\mathcal{M}_2$ sont déterministes, la relation de simulation $\mathcal{M}_2 \preceq_s \mathcal{M}_1$ est synonyme d'équivalence séquentielle.

1.2.2 Application à l'exemple

Le modèle $\mathcal{ARB} = (\mathcal{X}, \mathcal{O}, \mathcal{S}, \sigma_0, \mathcal{F}_t, \mathcal{F}_o)$ représente la machine d'états finis de la figure 1.1 :

$\mathcal{X} = \{000, 001, 010, 011, 100, 101, 110, 111\}$

$\mathcal{O} = \{000, 010, 100, 001\}$

$\mathcal{S} = \{000, 001, 010, 100\}$, $\sigma_0 = 000$

$\mathcal{F}_t = \{((000, 000), 000), ((000, 1--), 100), ((000, 01-), 010), ((000, 001), 001),$
$((100, 000), 000), ((100, 100), 100), ((100, -1-), 010), ((100, -01), 001),$
$((010, 000), 000), ((010, 010), 010), ((010, 0-1), 001), ((010, 1--), 100),$
$((001, 000), 000), ((001, 001), 001), ((001, 01-), 010), ((001, 1--), 100)\}$

Dans le cas particulier de cet arbitre, nous avons $\mathcal{F}_t = \mathcal{F}_o$.

1.3 Techniques de représentation

1.3.1 Représentation Booléenne

Dans l'exemple du paragraphe précédent, les états et les transitions sont représentés de manière *explicite*, par simple énumération. En pratique, ceci constitue un inconvénient important. La plupart des systèmes d'aujourd'hui, qu'ils soient décrits manuellement ou générés de manière automatique, ont un ensemble d'états de taille astronomique. Non seulement ces systèmes sont et seront impossibles à représenter de manière explicite, mais il sera également impossible de tirer quelque conclusion que ce soit concernant leur comportement.

La représentation *symbolique* constitue une approche alternative permettant d'aborder ce problème. Basée sur la *logique propositionnelle* de premier ordre, l'approche symbolique permet de manipuler non pas des états individuels mais des ensembles d'états. Construire une représentation symbolique d'une machine d'états finis $\mathcal{M}$ revient à réécrire l'ensemble de ses composants à l'aide de termes de la logique propositionnelle.

Tout terme de la logique propositionnelle est construit à partir d'un ensemble de *propositions atomiques* $\mathcal{P}$. Cet ensemble inclut les valeurs constantes *booléennes*, $\mathbb{B} = \{faux, vrai\}$ ainsi qu'un ensemble de variables prenant chacune des valeurs dans l'ensemble $\mathbb{B}$. Dans un souci de concision, les éléments de $\mathbb{B}$ seront associés aux valeurs entières "0" et "1". Un terme a donc une des formes suivantes :

- toute valeur de $\mathbb{B}$ est un terme ;
- toute autre proposition atomique $p \in \mathcal{P}$ est un terme ;
- si t_1 et t_2 sont des termes, alors $t_1.t_2$ et $t_1 + t_2$ sont également des termes. Les opérateurs "." et "+" dénotent respectivement la conjonction et la disjonction ;

– si t est un terme, alors $\bar{t}$, la négation de t est aussi un terme ;

Les opérateurs "implication" ($\Rightarrow$) et "équivalence" ($\Leftrightarrow$) sont dérivés des opérateurs logiques de base, employés ci-dessus. Ainsi, quels que soient les termes t_1 et t_2, l'expression $t_1 \Rightarrow t_2$ est identique à $\bar{t_1} + t_2$. De même, l'expression $t_1 \Leftrightarrow t_2$ est identique à $t_1.t_2 + \bar{t_1}.\bar{t_2}$.

La représentation symbolique d'une machine $\mathcal{M}$ passe par une étape préliminaire de construction d'un modèle booléen associé. Cette étape constitue un rapprochement du niveau conceptuel bi-valué de réalisation physique des circuits numériques actuels. On construit tout d'abord des correspondants booléens pour les ensembles $\mathcal{X}$, $\mathcal{O}$ et $\mathcal{S}$. Leurs éléments sont associés de manière injective avec des n-uplets formés sur $\mathbb{B}$. Pour que cette correspondance puisse être réalisée, la taille minimale de ces n-uplets doit être respectivement :

$$m = \lceil log_2|\mathcal{X}| \rceil \qquad p = \lceil log_2|\mathcal{O}| \rceil \qquad n = \lceil log_2|\mathcal{S}| \rceil$$

où $|\mathcal{X}|$, $|\mathcal{O}|$ et $|\mathcal{S}|$ dénotent les cardinaux des ensembles $\mathcal{X}$, $\mathcal{O}$ et $\mathcal{S}$, et $\lceil x \rceil$ dénote le plus petit nombre entier tel que le nombre réel $x \leq \lceil x \rceil$.

Ensuite, on définit les variantes binaires des fonctions $\mathcal{F}_t$ et $\mathcal{F}_o$. Dans la suite, les ensembles $X = \{x_1, x_2, ..., x_m\}$, $O = \{o_1, o_2, ..., o_p\}$ et $S = \{s_1, s_2, ..., s_n\}$ contiennent des propositions atomiques. Leurs éléments sont appelés respectivement variables booléennes d'entrée, de sortie et d'état. On note $\langle x_m \rangle = \langle x_1, x_2, ..., x_m \rangle$ le vecteur de taille m construit à partir des éléments de X. Les vecteurs $\langle o_p \rangle$ et $\langle s_n \rangle$ sont construits de façon similaire. Le vecteur $\langle x_m \rangle \bullet \langle s_n \rangle$ de taille $m + n$ est obtenu par concaténation des éléments des vecteurs $\langle x_m \rangle$ et $\langle s_n \rangle$.

Définition 1.11 (Modèle Booléen d'une machine d'états finis) *Soit* $\mathcal{M} = \langle \mathcal{X}, \mathcal{O}, \mathcal{S}, \sigma_0, \mathcal{F}_t, \mathcal{F}_o \rangle$ *une machine d'états finis.* $M = \langle X, O, S, s_0, F_t, F_o \rangle$ *est une représentation booléenne de* $\mathcal{M}$ *s'il existe trois fonctions d'encodage injectives* $E_X : \mathcal{X} \rightarrow \mathbb{B}^m$, $E_O : \mathcal{O} \rightarrow \mathbb{B}^p$, $E_S : \mathcal{S} \rightarrow \mathbb{B}^n$ *telles que*

- $s_0 = E_S(\sigma_0)$
- $F_t : \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}^n$ *fonction de transition vérifie :* $\forall \xi \in \mathcal{X}, \forall \sigma \in \mathcal{S}, F_t(E_X(\xi), E_S(\sigma)) = E_S(\mathcal{F}_t(\xi, \sigma))$;
- $F_o : \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}^p$ *fonction de sortie vérifie* $\forall \xi \in \mathcal{X}, \forall \sigma \in \mathcal{S}, F_o(E_X(\xi), E_S(\sigma)) = E_O(\mathcal{F}_o(\xi, \sigma))$.

Les fonctions d'encodage E_X, E_O et E_S sont injectives mais pas forcément surjectives, car on a les inégalités $|\mathcal{X}| \leq |\mathbb{B}^m|$, $|\mathcal{O}| \leq |\mathbb{B}^p|$ et $|\mathcal{S}| \leq |\mathbb{B}^n|$.

Soient X, O, S les images par les fonctions d'encodage des ensembles $\mathcal{X}$, $\mathcal{O}$ et $\mathcal{S}$. $X = E_X(\mathcal{X}) \subseteq \mathbb{B}^m, O = E_O(\mathcal{O}) \subseteq \mathbb{B}^p, S = E_S(\mathcal{S}) \subseteq \mathbb{B}^n$.

X, O et S ne sont la plupart du temps que des sous-ensembles booléens stricts de $\mathbb{B}^m$, $\mathbb{B}^n$ et $\mathbb{B}^p$. Cependant, il n'est pas réaliste de considérer qu'un circuit à m variables booléennes d'entrée accepte seulement un sous-ensemble de valeurs de $\mathbb{B}^m$. Une définition possible de F_t sur l'intégralité de l'ensemble $\mathbb{B}^m \times \mathbb{B}^n$ serait :

$$F_t(x, s) = \begin{cases} E_S(\mathcal{F}_t(\xi, \sigma)) & \text{si } \exists \xi, \sigma : x = E_X(\xi) \text{ et } s = E_S(\sigma) \\ s & \text{sinon} \end{cases}$$

Cette définition de F_t spécifie qu'aucun changement d'état ne se produit si x ou s n'ont pas d'antécédent dans $\mathcal{X}$ et $\mathcal{S}$. En ce qui concerne la fonction F_o, la solution généralement adoptée dans la pratique consiste à renvoyer une valeur "par défaut" pour toutes les combinaisons x et s non définies.

Moyennant l'emploi d'un langage de description adéquat, la redéfinition des fonctions de transition et de sortie s'effectue d'une manière naturelle : le concepteur laisse les transitions n'ayant pas de correspondant dans $\mathcal{M}$ non-spécifiées ; de façon similaire, une fonction de sortie renvoie une valeur par défaut choisie par le concepteur, sauf si ses arguments ont des antécédents dans $\mathcal{X}$ et $\mathcal{S}$.

Dans la suite, on emploiera exclusivement des fonctions de transition, et de sortie définies sur l'intégralité de $\mathbb{B}^m \times \mathbb{B}^n$.

1.3.2 Représentation symbolique

Dans la *représentation symbolique* de M, aussi bien les états individuels que les ensembles d'états sont exprimés grâce à leur fonction caractéristique.

Définition 1.12 (Représentation symbolique d'un état) *Soit $\sigma \in \mathcal{S}$ un état de la machine $\mathcal{M}$. Le codage de σ dans M, $E_S(\sigma) \in \mathbb{B}^n$ est un vecteur de n valeurs booléennes. La représentation symbolique de σ est donnée par la fonction caractéristique χ_σ :*

$$\chi_\sigma(\langle s_1, s_2, ..., s_n \rangle) = \prod_{i=1}^{n} (s_i \Leftrightarrow E_S^i(\sigma))$$

où E_S^i est la i-ème composante du vecteur E_S de taille n.

En manipulant les encodages de σ et σ' donnés par la fonction E_S, grâce aux vecteurs de variables d'état courant $\langle s_n \rangle$ et prochain état $\langle s'_n \rangle$, σ' est un successeur de σ s'écrit :

$$\langle s'_n \rangle = F_t(\langle x_m \rangle, \langle s_n \rangle) \tag{1}$$

soit en termes de la logique propositionnelle :

$$T(\langle x_m \rangle, \langle s_n \rangle, \langle s'_n \rangle) = \prod_{i=1}^{n} s'_i \Leftrightarrow F_t^i(\langle x_m \rangle, \langle s_n \rangle) \tag{2}$$

Le terme T défini par (2) est appelé une *relation de transition*. C'est une fonction caractéristique qui représente toutes les combinaisons légales dans M de valeurs sur les vecteurs d'entrée, état courant et prochain état.

Définition 1.13 (Modèle symbolique d'une machine d'états) *Le modèle symbolique d'une machine d'états $\mathcal{M}$ est un n-uplet $M = \langle X, O, S, T, F_o, \chi_0 \rangle$ contenant les ensembles de variables d'entrée, de sortie et d'état X, O et S, une relation de transition $T : \mathbb{B}^m \times \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$, une fonction de sortie $F_o : \mathbb{B}^m \times \mathbb{B}^n \to \mathbb{B}^p$ et une fonction caractéristique χ_0 qui représente l'état initial de M.*

Définition 1.14 (Machines d'états symboliques composables) *Soient* $M_A = \langle X_A, S_A, O_A, T_A, \chi_{0A} \rangle$ *et* $M_B = \langle X_B, S_B, O_B, T_B, \chi_{0B} \rangle$ *deux modèles symboliques. On dit que ces modèles sont composables si et seulement si* $O_A \cap O_B = \emptyset$.

Définition 1.15 (Produit synchronisé de deux machines d'états symboliques) *Soient* $M_A = \langle X_A, S_A, O_A, T_A, \chi_{0A} \rangle$ *et* $M_B = \langle X_B, S_B, O_B, T_B, \chi_{0B} \rangle$ *deux modèles symboliques composables. Le produit synchronisé de* M_A *avec* M_B[3]*, noté* $M_A || M_B$ *est la machine symbolique* $M_{prod} = \langle X_{prod}, S_{prod}, O_{prod}, T_{prod}, \chi_{0prod} \rangle$ *où* $X_{prod} \subseteq X_A \cup X_B$, $S_{prod} = S_A \cup S_B$, $O_{prod} \subseteq O_A \cup O_B$, $T_{prod} = T_A \,.\, T_B$ *et* $\chi_{0prod} = \chi_{0A} \cdot \chi_{0B}$.

Le modèle de machine d'états décrit à l'aide d'une relation de transition est connu sous le nom de *modèle de Kripke*. La relation de transition est construite de manière à regrouper tous les couples d'états adjacents. Cette caractéristique lui permet de représenter des systèmes non-déterministes, ce qui confère plus de généralité au modèle de Kripke. Dans l'équation (2), la relation de transition T est calculée à partir d'une fonction booléenne de transition. Elle correspond donc à un système déterministe. Cependant, le fait de se restreindre à l'étude des systèmes déterministes n'entraîne pas de perte de généralité. En effet, tout modèle non-déterministe peut être transformé en un modèle déterministe équivalent.

Propriétés de la représentation symbolique. L'emploi des fonctions caractéristiques permet de manipuler des ensembles arbitraires d'états d'une machine, quelle qu'en soit la cardinalité. Ainsi, toutes les opérations de manipulation des ensembles possèdent un correspondant booléen dans le domaine symbolique. La représentation symbolique d'un ensemble $\mathcal{E}$ d'états appartenant à une machine $\mathcal{M}$ satisfait les propriétés suivantes :

- $\chi_{\mathcal{E}}(\langle s_1, s_2, ..., s_n \rangle) = 0$ lorsque $\mathcal{E} = \emptyset$;
- l'union des ensembles $\mathcal{E}_1$ et $\mathcal{E}_2$ s'exprime par l'opération "ou" logique.
  $\chi_{\mathcal{E}_1 \cup \mathcal{E}_2}(\langle s_1, s_2, ..., s_n \rangle) = \chi_{\mathcal{E}_1}(\langle s_1, s_2, ..., s_n \rangle) + \chi_{\mathcal{E}_2}(\langle s_1, s_2, ..., s_n \rangle)$;
- l'intersection des ensembles $\mathcal{E}_1$ et $\mathcal{E}_2$ s'exprime par l'opération "et" logique.
  $\chi_{\mathcal{E}_1 \cap \mathcal{E}_2}(\langle s_1, s_2, ..., s_n \rangle) = \chi_{\mathcal{E}_1}(\langle s_1, s_2, ..., s_n \rangle).\chi_{\mathcal{E}_2}(\langle s_1, s_2, ..., s_n \rangle)$;
- le complément de l'ensemble $\mathcal{E}$ par rapport à $\mathcal{S}$ s'exprime par l'opération "non" logique : $\chi_{\mathcal{S} \setminus \mathcal{E}} = \chi_{\mathcal{S}} . \overline{\chi_{\mathcal{E}}}$. Dans le cas particulier où $|\mathcal{S}| = 2^n$, la fonction d'encodage $E_{\mathcal{S}}$ devient une bijection. Tout élément de $\mathbb{B}^n$ possède un antécédent dans $\mathcal{S}$. On a donc $\chi_{\mathcal{S}} = \chi_{\mathbb{B}^n} = 1$ et le complément de l'ensemble $\mathcal{E}$ devient $\chi_{\mathcal{S} \setminus \mathcal{E}} = \bar{\chi_{\mathcal{E}}}$.

On manipule délibérément l'état σ et l'ensemble contenant un unique élément $\{\sigma\}$ comme s'il s'agissait du même objet, en termes de représentation par fonction caractéristique.

Il existe une correspondance étroite entre le modèle symbolique d'une machine d'états finis et un circuit numérique séquentiel. Les variables d'entrée et de sortie du modèle correspondent aux ports entrants et sortants du circuit. Un modèle booléen met en œuvre des fonctions pour calculer les nouvelles valeurs des variables de sortie et d'état. Ces fonctions sont reproduites dans un circuit à l'aide de réseaux de portes logiques, réalisant chacune une fonction logique élémentaire. Quant aux variables d'état, elles ont pour correspondant électronique les mémoires (bascules).

[3] *aussi appelé composition parallèle de* M_A *et* M_B

Le temps étant considéré discret, toute transition (changement de valeur d'au moins une variable d'état) est déclenchée par les événements successifs produits par une *horloge*. Physiquement, un port entrant spécial intitulé *horloge* est connecté à toutes les bascules du circuit. Les événements produits sur le signal d'horloge déterminent le changement de valeur de toutes les mémoires.

Le modèle booléen d'une machine d'états constitue le point d'entrée dans les outils de vérification formelle. Cependant, pour des raisons pratiques liées à la manipulation de circuits réels, structurés ou non, il est utile d'étendre la définition d'un modèle booléen de façon à inclure les notions de *variable locale* et de *hiérarchie*. Le modèle hiérarchique de MEALY effectue cette extension.

1.3.3 Modèle hiérarchique de Mealy

Définition 1.16 (Modèle de Mealy hiérarchique) *Une machine hiérarchique de Mealy est un n-uplet* $MH = \langle X, O, S, L, \chi_0, F_t, F_o, F_L, COMP \rangle$*, où* X*,* O*,* S *sont respectivement des ensembles de variables d'entrée, de sortie, d'état. L'ensemble* L*, de taille* k*, contient les variables locales de la machine hiérarchique* MH*.* $\chi_0(s_1, ...s_n)$ *est la fonction caractéristique qui représente l'état initial de* MH*. La fonction* $F_t : \mathbb{B}^m \times \mathbb{B}^n \times \mathbb{B}^k \rightarrow \mathbb{B}^n$ *est la fonction de transition de* MH *:*

$$\langle s_1, s_2, ..., s_n \rangle = F_t(x_1, ..., x_m, s_1, ..., s_n, l_1, ...l_k). \tag{3}$$

La fonction $F_o : \mathbb{B}^m \times \mathbb{B}^n \times \mathbb{B}^k \rightarrow \mathbb{B}^p$ *est la fonction de sortie de* MH *:*

$$\langle o_1, o_2, ..., o_p \rangle = F_o(x_1, ..., x_m, s_1, ..., s_n, l_1, ...l_k). \tag{4}$$

La fonction $F_L : \mathbb{B}^m \times \mathbb{B}^n \times \mathbb{B}^k \rightarrow \mathbb{B}^k$ *est la fonction qui calcule les valeurs des variables locales de* MH *:*

$$\langle l_1, l_2, ...l_k \rangle = F_L(x_1, ..., x_m, s_1, ..., s_n, l_1, ...l_k). \tag{5}$$

$COMP = \{mh_1, mh_2, ..., mh_q\}$ *est un ensemble de taille* q *dont les éléments sont des machines* MH*.*

Les machines mh_i sont appelées des composants de MH. Les variables d'entrée de chaque composant mh_i correspondent obligatoirement à un sous-ensemble de $X \cup L$. De même, les variables de sortie des mh_i correspondent à un sous-ensemble de $L \cup O$. Par ailleurs, si $O_i, O_j \subset L \cup O$ sont les ensembles de sorties de deux composants mh_i et mh_j, alors $O_i \cap O_j = \emptyset$.

Toute machine hiérarchique peut être transformée en un modèle booléen en appliquant les étapes suivantes :

1. pour chaque mh_i dont l'ensemble $COMP$ est vide, remplacer dans les fonctions F_t et F_o, les variables locales l_i par leur fonction F_L^i correspondante ;
2. dans MH, remplacer tous les composants mh dont l'ensemble $COMP$ est vide par leur contenu. Puis, retour à l'étape 1.

Remarque. La définition du modèle hiérarchique de Mealy permet, grâce aux variables locales $l_1, ..., l_k$, de combiner certains composants mh_i et mh_j de façon à créer des dépendances fonctionnelles circulaires - boucles combinatoires - entre les variables de ces composants. Ce scénario n'est pas admissible dans le contexte de la modélisation des circuits numériques. Comme l'ajout de hiérarchie au sein d'une machine MH se fait manuellement, la vérification d'absence de boucles combinatoires doit être systématique.

1.3.4 Représentation symbolique de l'exemple

La représentation symbolique de la machine $\mathcal{ARB}$ passe par l'étape de construction d'un modèle booléen ARB. On calcule tout d'abord les cardinaux des ensembles de variables X, O et S :

$$m = |X| = \lceil log_2|\mathcal{X}| \rceil = 3$$
$$p = |O| = \lceil log_2|\mathcal{O}| \rceil = 2$$
$$n = |S| = \lceil log_2|\mathcal{S}| \rceil = 2$$

permettant de construire un codage booléen de $\mathcal{ARB}$ de taille minimale. Ainsi, $X = \{x_1, x_2, x_3\}$, $S = \{s_1, s_2\}$ et $O = \{o_1, o_2\}$. Un choix possible des fonctions d'encodage E_X, E_S et E_O serait le suivant :

$$E_X = \{(000, \langle 000\rangle), (001, \langle 001\rangle), (010, \langle 010\rangle),$$
$$(011, \langle 011\rangle), (100, \langle 100\rangle), (101, \langle 101\rangle), (110, \langle 110\rangle), (111, \langle 111\rangle)\}$$
$$E_S = \{(000, \langle 00\rangle), (001, \langle 01\rangle), (010, \langle 10\rangle), (100, \langle 11\rangle)\}$$
$$E_O = E_S$$

où $\langle 00\rangle$ représente le codage booléen de l'état initial $\sigma_0 = 000$ de M_{arb}. Les fonctions F_t et F_o se déduisent à partir de $\mathcal{F}_t$ et $\mathcal{F}_o$ et E_S :

$$F_t = \{((\langle 00\rangle, \langle 000\rangle), \langle 00\rangle), ((\langle 00\rangle, \langle 1--\rangle), \langle 11\rangle), ((\langle 00\rangle, \langle 01-\rangle), \langle 10\rangle),$$
$$((\langle 00\rangle, \langle 001\rangle), \langle 01\rangle), ((\langle 11\rangle, \langle 000\rangle), \langle 00\rangle), ((\langle 11\rangle, \langle 100\rangle), \langle 11\rangle),$$
$$((\langle 11\rangle, \langle -1-\rangle), \langle 10\rangle), ((\langle 11\rangle, \langle -01\rangle), \langle 01\rangle), ((\langle 10\rangle, \langle 000\rangle), \langle 00\rangle),$$
$$((\langle 10\rangle, \langle 010\rangle), \langle 10\rangle), ((\langle 10\rangle, \langle 0-1\rangle), \langle 01\rangle), ((\langle 10\rangle, \langle 1--\rangle), \langle 11\rangle),$$
$$((\langle 01\rangle, \langle 000\rangle), \langle 00\rangle), ((\langle 01\rangle, \langle 001\rangle), \langle 01\rangle), ((\langle 01\rangle, \langle 01-\rangle), \langle 10\rangle),$$
$$((\langle 01\rangle, \langle 1--\rangle), \langle 11\rangle)\}$$

La fonction F_t est en fait un vecteur de deux fonctions $\langle F_t^1, F_t^2\rangle$. Leur expression booléenne se déduit directement de la description relationnelle de F_t :

$$F_t^1(s_1, s_2, x_1, x_2, x_3) = \bar{s}_1.\bar{s}_2.x_1 + \bar{s}_1.\bar{s}_2.\bar{x}_1.x_2 + s_1.s_2.x_1.\bar{x}_2.\bar{x}_3 + s_1.s_2.x_2 +$$
$$s_1.\bar{s}_2.\bar{x}_1.x_2.\bar{x}_3 + s_1.\bar{s}_2.x_1 + \bar{s}_1.s_2.\bar{x}_1.x_2 + \bar{s}_1.s_2.x_1$$
$$F_t^2(s_1, s_2, x_1, x_2, x_3) = \bar{s}_1.\bar{s}_2.x_1 + \bar{s}_1.\bar{s}_2.\bar{x}_1.\bar{x}_2.\bar{x}_3 + s_1.s_2.x_1.\bar{x}_2.\bar{x}_3 + s_1.s_2.\bar{x}_2.x_3 +$$
$$s_1.\bar{s}_2.\bar{x}_1.x_3 + s_1.\bar{s}_2.x_1 + \bar{s}_1.s_2.\bar{x}_1.\bar{x}_2.x_3 + \bar{s}_1.s_2.x_1$$

Les termes symboliques représentant les états de ARB sont déterminés par les fonctions caractéristiques :

$$\chi_{\langle 00\rangle}(s_1, s_2) = \bar{s}_1.\bar{s}_2 \quad \chi_{\langle 01\rangle}(s_1, s_2) = \bar{s}_1.s_2 \quad \chi_{\langle 10\rangle}(s_1, s_2) = s_1.\bar{s}_2 \quad \chi_{\langle 11\rangle}(s_1, s_2) = s_1.s_2$$

conformément à la définition 1.12. Le prédicat représentant l'état initial de ARB est $\chi_0(s_1, s_2) = \bar{s}_1.\bar{s}_2$.

Soit $S' = \{s'_1, s'_2\}$ l'ensemble de variables permettant de désigner les états successeurs dans ARB. Par application de l'équation 2, la représentation propositionnelle de la relation de transition T de ARB :

$$T = (s'_1 \Leftrightarrow F_t^1(s_1, s_2, x_1, x_2, x_3)).(s'_2 \Leftrightarrow F_t^2(s_1, s_2, x_1, x_2, x_3))$$

La fonction de sortie de ARB est $F_o = F_t$.

Remarque. En réalité, le codage minimal choisi pour les valeurs de sortie de ARB peut se révéler incommode pour le processus de conception. Le système de l'exemple met en œuvre une fonctionnalité d'arbitrage pour trois clients distincts. Chaque client communique à l'arbitre sa requête, par l'intermédiaire d'une variable d'entrée $x_i \in X$. A chaque instant t, l'arbitre analyse l'ensemble des trois requêtes et décide de donner l'accès à un seul parmi ses clients. Cette réponse est donnée à travers les variables de sortie o_1 et o_2. :

- $\langle o_1, o_2 \rangle = \langle 0, 0 \rangle$: aucun client n'a d'autorisation d'accès ;
- $\langle o_1, o_2 \rangle = \langle 0, 1 \rangle$, $\langle o_1, o_2 \rangle = \langle 1, 0 \rangle$ ou $\langle o_1, o_2 \rangle = \langle 1, 1 \rangle$: autorisation d'accès à l'instant t accordée pour le client 3 ou 2 ou 1.

Ainsi, chaque client doit lire et décoder les valeurs o_1 et o_2 pour déterminer si une autorisation lui a été accordée. Cependant, une solution plus pratique et élégante aurait consisté à gérer séparément les autorisations d'accès, en utilisant une variable d'état et de sortie par client. La machine ARB' pour cette solution alternative aurait un ensemble de variables d'état $S = \{s_1, s_2, s_3\}$ et de sortie $O = \{o_1, o_2, o_3\}$ telles qu'à tout instant, une et une seule variable d'état et de sortie aurait la valeur 1. Dans ce cas, on construit trois fonctions de transition à partir de la description de $\mathcal{ARB}$: les symboles des alphabets d'état $\mathcal{S}$ et de sortie $\mathcal{O}$ peuvent être reconstitués par concaténation des valeurs possibles de s_1, s_2, s_3 et o_1, o_2, o_3 Compte-tenu du fait que seules quatre combinaisons sur s_1, s_2 et s_3 sont acceptables, conformément à l'ensemble $\mathcal{S}$, les fonctions de transition F_t^1, F_t^2 et F_t^3 peuvent être simplifiées :

$$F_t^1(s_1, s_2, s_3, x_1, x_2, x_3) = x_1.(\bar{s}_1 + s_1.\bar{x}_2.\bar{x}_3)$$
$$F_t^2(s_1, s_2, s_3, x_1, x_2, x_3) = x_2.(\bar{s}_2.(\bar{x}_1 + s_1) + s_2.\bar{x}_1.\bar{x}_3)$$
$$F_t^3(s_1, s_2, s_3, x_1, x_2, x_3) = x_3.(\bar{s}_3.(\bar{x}_2 + s_2).(\bar{x}_1 + s_1) + s_3.\bar{x}_1.\bar{x}_2)$$

Les termes symboliques représentant les états de ARB' sont déterminés par les fonctions caractéristiques :

$$\chi_{\langle 000 \rangle}(s_1, s_2, s_3) = \bar{s}_1.\bar{s}_2.\bar{s}_3 \quad \chi_{\langle 100 \rangle}(s_1, s_2, s_3) = \bar{s}_1.\bar{s}_2.\bar{s}_3$$
$$\chi_{\langle 010 \rangle}(s_1, s_2, s_3) = \bar{s}_1.s_2.\bar{s}_3 \quad \chi_{\langle 001 \rangle}(s_1, s_2, s_3) = \bar{s}_1.\bar{s}_2.s_3$$

Le prédicat représentant l'état initial de ARB est $\chi_0(s_1, s_2, s_3) = \bar{s}_1.\bar{s}_2.\bar{s}_3$ et les fonctions de sortie de ARB' sont identiques aux fonctions de transition.

1.4 Parcours symbolique d'une machine d'états finis

Le modèle symbolique d'une machine d'états sert de point de départ pour la preuve de propriétés dans les outils actuels de vérification de modèles. La preuve que le modèle M satisfait la propriété P s'appuie sur quelques opérations de base, exprimées sur des termes de la logique propositionnelle. Typiquement, on calcule l'ensemble d'états de M (exprimé par une fonction caractéristique) dans lesquels P est vraie. Ce calcul se fait de manière

incrémentale avec une granularité égale à une transition dans M. Les opérations de base sont le calcul de *l'image* et la *pré-image* d'un ensemble d'états.

La mise en œuvre des opérations d'image et de pré-image utilise les fonctions caractéristiques de la relation de transition du modèle symbolique et de l'ensemble d'états de départ. Leur efficacité repose donc sur la facilité de représentation, de manipulation et d'évaluation de ces fonctions. D'autre part, il s'avère que la construction de la relation de transition T à partir d'un circuit industriel est une opération bien trop coûteuse. En pratique, il semble plus réaliste de partitionner T en plusieurs sous-produits, ou bien d'en garder une représentation implicite, sous forme d'un ensemble de relations atomiques $s'_i \Leftrightarrow F^i_t(\langle x_m \rangle, \langle s_n \rangle), \forall i = 1..n$. Il s'agit des approches *relationnelle* ou *fonctionnelle* pour le calcul d'image.

Cette section passe en revue les définitions de l'image et de la pré-image, ainsi que les aspects liés à la représentation et l'évaluation des ensembles d'états durant ce calcul. Suivra une présentation de l'état de l'art des approches relationnelle et fonctionnelle du calcul d'image.

1.4.1 Image et pré-image d'un ensemble d'états

Définition 1.17 (Image d'une fonction booléenne) *Soit $f_m : \mathbb{B}^n \rightarrow \mathbb{B}^m$ une fonction vectorielle à m composantes, dépendant chacune de n variables booléennes, $\langle y_m \rangle = f_m(\langle x_n \rangle)$, où $\langle y_m \rangle$ et $\langle x_n \rangle$ sont des vecteurs de variables de taille m et n. L'image $\textsc{Img}_{f_m}$ de f_m est le sous-ensemble de tous les vecteurs booléens de taille m que f_m peut produire. La fonction caractéristique de ce sous-ensemble est donnée par :*

$$\chi_{\textsc{Img}_{f_m}}(y_1, y_2, ..., y_m) = \exists x_1, x_2, ..., x_n : \prod_{i=1}^{m} (y_i \Leftrightarrow f^i_m(\langle x_n \rangle))$$

Dans le cadre du parcours symbolique d'une machine d'états M, la notion d'image s'applique au calcul des successeurs d'un ensemble d'états. On parle alors d'image d'un ensemble d'états E à travers la relation de transition T de M :

Définition 1.18 (Image et pré-image d'un ensemble d'états) *Soit M le modèle symbolique d'une machine d'états finis. Soit E un sous-ensemble de $\mathbb{B}^n$, représenté par sa fonction caractéristique χ_E. L'image $\textsc{Img}(\chi_E, T)$ de E à travers la relation de transition T de M est un sous-ensemble de $\mathbb{B}^n$ dont la fonction caractéristique est :*

$$\chi_{\textsc{Img}(\chi_E,T)}(\langle s'_n \rangle) = \exists \langle x_m \rangle \in \mathbb{B}^m, \exists \langle s_n \rangle \in \mathbb{B}^n : T(\langle x_m \rangle, \langle s_n \rangle, \langle s'_n \rangle).\chi_E(\langle s_n \rangle) \qquad (6)$$

La pré-image est l'opération duale de l'image qui permet de calculer l'ensemble $\textsc{PreImg}(\chi_E, T)$ des états prédécesseurs de E :

$$\chi_{\textsc{PreImg}(\chi_E,T)}(\langle s_n \rangle) = \exists \langle x_m \rangle \in \mathbb{B}^m, \exists \langle s'_n \rangle \in \mathbb{B}^n : T(\langle x_m \rangle, \langle s_n \rangle, \langle s'_n \rangle).\chi_E(\langle s'_n \rangle) \qquad (7)$$

Par application successive de calculs d'image on parcourt progressivement l'ensemble des états atteignables de M. Ce calcul démarre à partir de son état initial. Soit $S_0 = \{\langle s_0 \rangle\} \subset$

$\mathbb{B}^n$ l'ensemble contenant l'état initial de M. A chaque pas i on calcule χ_{S_i}, la fonction caractéristique de l'ensemble d'états S_i accessibles à partir de s_0 en au plus i transitions :

$$\chi_{S_{i+1}} = \chi_{S_i} + \chi_{\text{IMG}(\chi_{S_i},T)} \tag{8}$$

La suite $S_0, S_1, ..., S_i, ...$ est croissante. Comme M contient un nombre fini d'états, ($S_i \subset \mathbb{B}^n$ quelle soit la valeur de i) cette suite converge. En d'autres termes, il existe une valeur k à partir de laquelle les états successeurs de S_k sont tous inclus dans S_k : $\text{IMG}(S_k, T) \subseteq S_k$. Donc, l'équation (8) admet un point fixe. Cette solution est la fonction caractéristique χ_{S_k} représentant tous les états de M pouvant être atteints à partir de $\langle s_0 \rangle$.

Définition 1.19 (Invariant) *Si les états atteignables S_k de M satisfont la propriété P, alors $\chi_{S_k} \Rightarrow P$. On dit que P est un* invariant *dans M.*

1.4.2 Ensembles d'états : représentation et évaluation

Les opérations d'image et de pré-image sont la clef de voûte de la technique de vérification symbolique de modèles. L'efficacité de leur mise en oeuvre constitue encore aujourd'hui l'objet de nombreux travaux de recherche. L'image en tant que représentation propositionnelle d'un ensemble d'états peut être manipulée comme une fonction booléenne. En réalité, lorsque le modèle parcouru est extrait d'un circuit industriel, cette fonction dépend d'un nombre très grand de variables et son expression est souvent extrêmement complexe. Par ailleurs, la grande majorité des problèmes de vérification de modèles nécessitent un calcul itératif de point fixe, enchaînant plusieurs calculs d'image. La condition d'arrêt d'un algorithme de point fixe s'exprime sur des ensembles d'états : si $S_i = S_{i+1}$ alors un point fixe a été atteint et le calcul peut s'arrêter. Etant donné que $S_i \subseteq S_{i+1}$, cette même condition peut se transcrire sous la forme $S_{i+1} \setminus S_i = \emptyset$, où en termes de fonctions caractéristiques :

$$\chi_{S_{i+1}} \cdot \bar{\chi}_{S_i} = 0 \tag{9}$$

Dans ce contexte, deux problèmes très importants se posent lors du parcours symbolique d'un modèle :

- construire une représentation booléenne compacte des fonctions caractéristiques χ_{S_i} ;
- trouver, si possible, au moins une combinaison de valeurs montrant que l'équation (9) n'est pas une tautologie. Dans le cas contraire, un point fixe a été atteint. Ce problème est connu sous le nom de SAT. Dans la théorie de la complexité des algorithmes il sert de base pour la définition de la classe des problèmes NP-complets.

Dans le cadre du calcul symbolique d'image, deux principales directions ont été explorées pour la représentation et l'évaluation des fonctions booléennes.

1.4.2.1 Les Diagrammes de Décision Binaire

Le parcours symbolique a tout d'abord été basé sur l'emploi des *Diagrammes de Décision Binaire* (BDDs[4]) pour représenter des fonctions caractéristiques [63]. Introduit par

[4] en anglais : *Binary Decision Diagram*

Akers [6], ce concept s'appuie sur la décomposition de Boole appliquée à une fonction booléenne f. Par la suite, Bryant combine l'application récursive de la décomposition et le regroupement des sous-arbres isomorphes générés. Le résultat est un graphe acyclique dont les noeuds sont étiquetés avec les noms des variables de f. Ce graphe comporte deux noeuds *terminaux*, étiquetés "0" et "1" [15]. Bryant montre également l'importance de l'ordre de variables selon lesquelles la décomposition de Boole a lieu, pour la concision de la représentation résultante. Cette représentation est intitulée ROBDD (BDD réduit et ordonné). Pour un ordre de variables préalablement choisi, la représentation par ROBDDs est canonique. Ainsi, chaque fonction booléenne a une représentation canonique OBDD. De même, toute opération logique entre deux fonctions booléennes peut s'exprimer sur les ROBDDs correspondants, le résultat étant également un ROBDD. De nombreux paquetages très performants existent, tels que [51, 79], mettant en œuvre des primitives pour la construction et la manipulation efficace des BDDs.

Les diagrammes de décision binaire apportent à la fois une solution de représentation et d'évaluation des fonctions booléennes : la représentation souvent compacte d'une fonction s'accompagne également d'une réponse au problème SAT appliqué à cette fonction. La complexité spatiale de la représentation par BDDs d'une fonction est au pire des cas exponentielle dans le nombre de ses variables. Cependant, dans beaucoup d'applications pratiques la taille des BDDs obtenus se trouve bien en dessous de cette limite exponentielle.

1.4.2.2 Les techniques de résolution SAT

La découverte récente de nouvelles heuristiques pour la résolution de SAT [84, 81, 43] remet en question avec un certain succès l'emploi des BDDs pour le parcours symbolique d'une machine d'états.

L'approche de la *vérification de modèles bornés* (BMC[5]) [9] fait appel à un moteur de résolution SAT dans le but de trouver un contre-exemple montrant qu'une propriété P est fausse. Cette technique construit une suite M_i d'approximations de M aux i premières transitions possibles à partir de son état initial. Soit T la relation de transition de M :

$$\begin{aligned} M_0 &= \chi_0 \text{ - l'état initial de } M \\ M_i &= M_{i-1} \,.\, T(\langle x_m^{i-1}\rangle, \langle s_n^{i-1}\rangle, \langle s_n^i\rangle) \text{ - pour } i = 0..k \end{aligned}$$

La construction de M_i revient à *dérouler* T sur i pas. La relation de transition T peut être déroulée au plus k fois, où k est un nombre entier quelconque, ne dépassant pas le *diamètre*[6] du graphe de transition de M.

La technique BMC procède de manière itérative, en construisant progressivement les modèles M_i et en évaluant SAT$(M_i \,.\, \bar{P})$ pour $i = 0, ..., k$. Le processus s'arrête dès que $i = k$ ou dès que la réponse à cette évaluation est positive : le moteur de résolution SAT a trouvé un ensemble de valeurs pour les variables d'entrée $\langle x_m^0\rangle, ...\langle x_m^{i-1}\rangle$ qui représente une séquence de i transitions menant à un état où $\bar{P}$ est vraie. Cette solution constitue le contre-exemple qui appuie l'affirmation "P est fausse dans M_i et donc fausse dans M".

[5] en anglais : *Bounded Model Checking*

[6] mesure égale au nombre de pas de calcul de l'ensemble d'états atteignables de M.

Dans [2], Abdulla et al. proposent une approche de parcours symbolique utilisant également SAT. Le circuit est représenté comme un ensemble de fonctions booléennes. Un algorithme d'optimisation permet de minimiser la taille de ces fonctions, en se basant sur la réutilisation des sous-expressions. Le calcul de l'image impliquant une quantification existentielle sur des variables d'entrée et d'état est remplacé par un calcul itératif dont le résultat est une nouvelle formule booléenne optimisée. Un moteur de résolution SAT est appelé pendant le parcours symbolique lorsque l'évaluation d'une équation similaire à (9) est nécessaire, pour déterminer si un point fixe a été atteint.

1.4.3 Mise en œuvre

Compte-tenu de leur complexité, le choix entre BDDs et SAT se base avant tout sur le type de ressource que l'on souhaite utiliser de façon intensive : mémoire ou processeur. En pratique il est très difficile de raffiner ce critère. Mais l'expérience montre que la performance de ces techniques varie beaucoup selon la nature du problème à traiter et la structure des fonctions booléennes manipulées. Toutefois, à cause de la complexité des circuits, pour lesquels ne serait-ce que le nombre de ports entrants se comptent par centaines, l'utilisation directe de SAT ou des BDDs a peu de chances d'aboutir. En particulier, construire la représentation par BDDs d'une relation de transition en vue d'un calcul d'image est une opération quasi-impossible sur un circuit dépassant la taille et la complexité d'un exemple scolaire.

La *décomposition disjonctive* par application du théorème de Boole constitue une heuristique de base, permettant d'améliorer aussi bien l'efficacité de la représentation par BDDs, que celle de la résolution de SAT.

Application à la manipulation des BDDs L'expérience montre que des pics d'utilisation de la mémoire dûs à l'explosion de la taille des BDDs se produisent *pendant* le calcul d'image, alors qu'en général le résultat de ce calcul possède une représentation BDD très compacte. En ce sens, la décomposition disjonctive des fonctions booléennes complexes peut se montrer utile. De cette manière, chaque opération booléenne peut se réécrire grâce à un nombre de pas intermédiaires dont les opérandes sont des cofacteurs, de taille plus modérée.

Le principe de la décomposition disjonctive est le suivant. Pour une fonction booléenne $f(\langle x_n \rangle)$, on choisit un sous-ensemble V_k de k parmi ses n variables. On note $C_i, i = 1..2^k$ les termes représentant tous les produits possibles entre les littéraux (complémentés ou non) contenus dans V_k. On emploie également la notion de cofacteur de la fonction f par rapport à un ensemble de littéraux, complémentés ou non, désignés par C_i : il est noté $f|_{C_i}$.

Exemple. Soit $T(\langle x_3 \rangle, \langle s_2 \rangle, \langle s'_2 \rangle)$ la relation de transition du modèle symbolique ARB. Si $k = 2$ et $V_2 = \{s_1, s'_1\}$, alors quatre produits peuvent être formés sur les éléments de V_2 :

$$C_1 = s_1.s'_1, \quad C_2 = s_1.\bar{s}'_1, \quad C_3 = \bar{s}_1.s'_1 \quad \text{et } C_4 = \bar{s}_1.\bar{s}'_1$$

De même, les quatre cofacteurs de T_{arb} par rapport aux variables contenues dans X_2 sont :

$T_{arb}\big|_{C_1} = T_{arb}\big|_{s_1 s'_1}$, $T_{arb}\big|_{C_2} = T_{arb}\big|_{s_1 \bar{s}'_1}$, $T_{arb}\big|_{C_3} = T_{arb}\big|_{\bar{s}_1 s'_1}$, et $T_{arb}\big|_{C_4} = T_{arb}\big|_{\bar{s}_1 \bar{s}'_1}$

□

Par application du théorème de décomposition de Boole par rapport aux variables de V_k, la fonction f s'écrit :

$$f(\langle x_n \rangle) = \sum_{i=1}^{2^k} (C_i . f\big|_{C_i})$$

Chacun des 2^k composants de f aura une représentation BDD de taille strictement inférieure à celle de f. Ainsi, une opération booléenne binaire en termes de BDDs $Op \in \{.\ +\}$, entre f et une fonction booléenne arbitraire g peut s'effectuer progressivement, par construction successive des composantes $f\big|_{C_i}$ et tout en conservant des structures BDD de taille modérée. L'algorithme suivant décrit ces étapes de calcul :

Algorithme : calculer f Op g

1. $i \leftarrow 1$; $BDDresultat \leftarrow 0$;
2. construire le BDD de g ;
3. construire le BDD de $C_i . f\big|_{C_i}$;
4. construire le BDD de $(C_i . f\big|_{C_i})$ Op g ;
5. $BDDresultat \leftarrow BDDresultat + (C_i . f\big|_{C_i})$ Op g ;
6. si $i < n$ alors $i \leftarrow i + 1$ et retour au pas 3 ;

Cet algorithme permet de construire la représentation par BDD de f Op g tout en évitant de construire le BDD de f. En pratique, le choix judicieux de l'ensemble V_k peut entraîner des simplifications très importantes de certains composants parmi $f\big|_{C_i}$.

Bien qu'applicable à toute opération booléenne, cette heuristique se montre particulièrement efficace dans le cas du calcul d'image. Dans ce cas $Op = \{\ .\ \}$, $f = T$ et $g = \chi_E$. L'expression de l'image (avant la quantification existentielle) devient :

$$\sum_{i=1}^{2^k} (C_i . T\big|_{C_i})\ Op\ \chi_E = \sum_{i=1}^{2^k} (\chi_E\ .\ C_i\ .\ T\big|_{C_i})$$

Examinons le calcul du BDD correspondant à chaque produit $\chi_E\ .\ C_i\ .\ T\big|_{C_i}$ dans cette somme. L'ordre d'évaluation des termes de chacun de ces produits est important. Le terme $C_i . T\big|_{C_i}$ aura toujours une représentation BDD de taille strictement plus grande que $T\big|_{C_i}$. En revanche il peut être intéressant d'évaluer en priorité les sous-expressions $\chi_E\ .\ C_i$, dont le résultat peut avoir une représentation BDD très simple.

Les avantages de cette décomposition sont illustrés sur le modèle ARB de l'arbitre donné en exemple dans ce chapitre. On calcule la représentation par BDDs de l'image de l'état initial χ_0 de ARB à travers sa relation de transition T. Soit $V_2 = \{s_1, s'_1\}$ un sous-ensemble de variables de T. Pour ce calcul d'image, au lieu de construire le BDD de T, on construit progressivement, l'ensemble de ses quatre cofacteurs par rapport aux éléments de V_2. Par application du théorème de Boole et en évaluant en priorité les sous-expressions $C_i\ .\ \chi_0$ l'expression de l'image de χ_0 à travers T s'écrit :

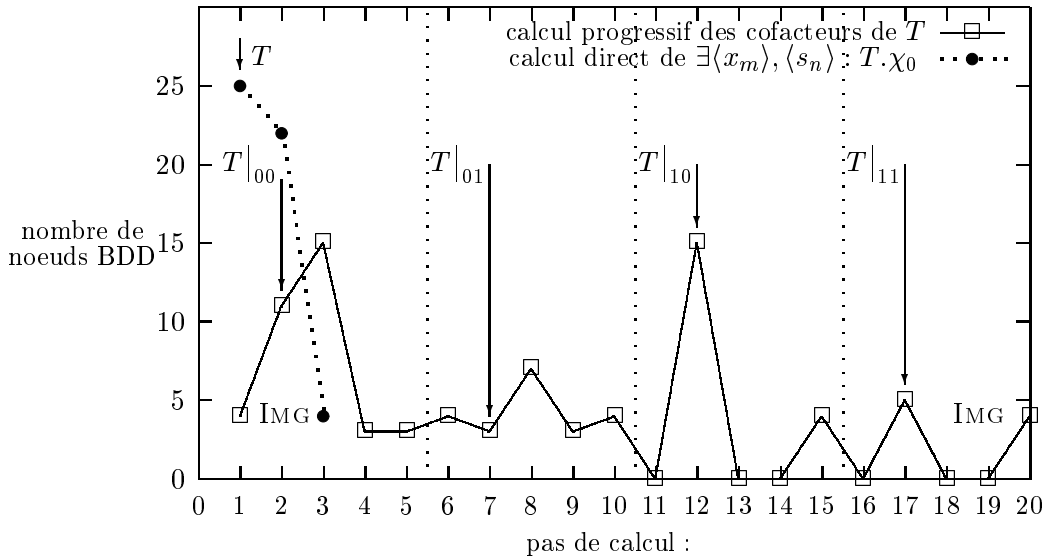

FIG. 1.2 – Tailles des BDDs dans le calcul de $\text{IMG}(T_{arb}, \chi_0)$

$$\text{IMG}(\chi_0, T) = \exists\langle x_m\rangle \in \mathbb{B}^m, \exists\langle s_n\rangle \in \mathbb{B}^n :$$
$$((\chi_0.\bar{s}_1.\bar{s}'_1).T\big|_{\bar{s}_1\bar{s}'_1} + (\chi_0.\bar{s}_1.s'_1).T\big|_{\bar{s}_1 s'_1} + (\chi_0.s_1.\bar{s}'_1).T\big|_{s_1\bar{s}'_1} + (\chi_0.s_1.s'_1).T\big|_{s_1 s'_1})$$

L'algorithme *calculer f Op g*, calcule les quatre termes de cette expression booléenne. Les courbes de la figure 1.2 illustrent (à une échelle extrêmement réduite) l'utilité de la décomposition, en termes d'utilisation de la mémoire (nombre de nœuds BDD) dans le cas du calcul d'image. La courbe en pointillés représente le calcul direct de l'image, décomposé en trois pas :

1. calcul de la relation de transition T ;
2. calcul du terme $T.\chi_0$;
3. quantification existentielle des variables $\langle x_m\rangle$ et $\langle s_n\rangle$.

La courbe continue représente le même calcul d'image qui utilise les cofacteurs de T. Ce calcul se déroule en quatre étapes, comportant chacune les opérations suivantes :

1. calcul du terme $\chi_0 \,.\, C_i$;
2. calcul du terme $T\big|_{C_i}$;
3. calcul du terme $(\chi_0 \,.\, C_i) \,.\, T\big|_{C_i}$;
4. quantification existentielle des variables $\langle x_m\rangle$ et $\langle s_n\rangle$;
5. calcul du BDD résultat (pas 5 de l'algorithme).

On remarque que lorsque le terme $\chi_0 \,.\, C_i$ est nul, il n'est pas nécessaire d'effectuer le pas 2 en calculant $T\big|_{C_i}$. Le graphique montre que l'emploi de la décomposition de Boole permet une meilleure utilisation de la mémoire que le calcul direct.

La méthode *fonctionnelle* de calcul d'image, dont il sera fait état dans la prochaine section, est axée sur l'emploi de cette heuristique.

Application à la résolution de SAT De très nombreuses approches de résolution de SAT [67, 43, 52] se basent sur la méthode de recherche arborescente utilisant la *décomposition des variables*[7]. Cette méthode emploie de manière intensive la décomposition de Boole ; la recherche d'une solution satisfaisant la fonction f se fait en trois étapes :

- choix d'une variable x_j pour appliquer la décomposition de Boole ;
- calcul des cofacteurs $f\big|_{x_i}$ et $f\big|_{\bar{x}_i}$ de la fonction f selon la variable x_i. Cette étape est également appelée *propagation des contraintes Booléennes*. Un des deux cofacteurs peut se révéler être de la forme $0.g$. Cette expression n'est pas satisfaisable et donc éliminée du champ de recherche ;
- recherche d'une solution satisfaisant $f\big|_{x_i}$ ou $f\big|_{\bar{x}_i}$.

Comme dans le cas de la construction des BDDs, l'efficacité de cette méthode dépend de l'ordre des variables selon lequel s'effectue la recherche d'une solution. Les moteurs actuels de résolution SAT cherchent à choisir d'abord les variables pouvant engendrer des cofacteurs de f les plus simples possibles. Ainsi, si f est une forme normale disjonctive, elle est décomposée selon la variable x_i à condition que dans les cofacteurs $f\big|_{x_i}$ et $f\big|_{\bar{x}_i}$ obtenus un maximum de clauses puissent se transformer en contradictions, grâce aux contraintes $x_i = 0$ ou $x_i = 1$, et être éliminées de la recherche.

L'approche de décomposition disjonctive appliquée aux BDDs présente également des avantages dans le cas de la résolution de SAT. Pour une fonction booléenne $f(\langle x_n \rangle)$, après sélection d'un ensemble X_k de variables et construction des 2^k cofacteurs de f selon les variables de X_k, le problème $\text{SAT}(f)$ se trouve décomposé en autant de sous-problèmes $\text{SAT}(f\big|_{C_i})$, $i = 1..2^k$, indépendants. Chaque sous-problème a une complexité $O(2^{n-k})$ et peut être résolu indépendamment des autres. Grâce à cette qualité, le problème $\text{SAT}(f)$ est bien plus facile à résoudre que la construction d'un BDD représentant f. Cependant, l'expérience montre que dans le cadre du parcours symbolique pour calculer un point fixe, l'approche SAT peut rencontrer des problèmes liés à la taille des formules booléennes générées. A chaque itération, la complexité de ces formules augmente, car un calcul d'image se traduit par une série d'opérations "ou" logique, engendrées par la quantification existentielle. L'optimisation de la formule booléenne résultante peut se révéler moins efficace que l'opération de réduction utilisée lors de la construction d'un BDD. En revanche, l'approche BMC travaille sur des modèles bornés et donne des résultats prometteurs sur des modèles difficiles à manipuler par BDDs tels que les multiplieurs.

1.4.4 Retour sur le calcul de l'image : relationnel ou fonctionnel ?

En pratique, la construction d'un BDD représentant la relation de transition T d'un modèle constitue une opération extrêmement coûteuse. Ce BDD, obtenu en effectuant explicitement les conjonctions de l'équation (2) aurait une taille très grande, ce qui entraînerait des conséquences sur l'efficacité du calcul d'image.

Au lieu de construire une représentation *monolithique* de la relation de transition, il est plus efficace de stocker l'ensemble de ses composants $T_i = s'_i \Leftrightarrow F^i_t(\langle x_m \rangle, \langle s_n \rangle), i = 1..n$ et de considérer que le produit des T_i reste implicite. Cette approche, basée sur la décomposition, a donné naissance à deux méthodes de calcul de l'image : relationnelle et

[7] en anglais : variable splitting

fonctionnelle.

1.4.4.1 La méthode relationnelle

A partir d'une relation de transition décomposée, la mise en œuvre du calcul d'image n'est pas triviale : le calcul ne peut pas être effectué de manière progressive, en quantifiant un composant T_i à la fois, à cause du fait que la quantification existentielle n'est pas distributive sur l'opération "et" logique. Cependant, lorsque l'on effectue la quantification existentielle de la variable $v_i \in X \cup S$, toute sous-formule qui ne dépend pas de v_i peut être déplacée en dehors de l'opération de quantification. Cette observation peut être exploitée dans deux directions, selon les propriétés structurelles du modèle parcouru : la *quantification "au plus tôt"* [53, 27] et la *partition de la relation de transition* [74].

La quantification "au plus tôt". Cette approche a pour but de calculer l'image d'un ensemble d'états, tout en évitant la construction exhaustive de la relation de transition. A partir des composants T_i et de la fonction caractéristique de l'ensemble de départ χ_E, on essaie d'alterner autant que possible les opérations "et logique" et "quantification existentielle". Pour cela, on réordonne les composants T_i de la relation de transition, en se basant sur leur dépendance par rapport aux variables de $X \cup S$. Soit $\rho : \{1..n\} \rightarrow \{1..n\}$ une permutation. La formule de calcul d'image de χ_E à travers T peut se réécrire :

$$\chi_{\text{IMG}(T,\chi_E)} = \exists \langle x_m \rangle \in \mathbb{B}^m, \exists \langle s_n \rangle \in \mathbb{B}^n : \prod_{i=1}^{n} T_{\rho(i)}(\langle x_m \rangle, \langle s_n \rangle, \langle s'_n \rangle) . \chi_E(\langle s_n \rangle) \qquad (10)$$

Quelle que soit la permutation ρ, il est possible de construire les sous-ensembles $Q_i, i = 1..n$ de $X \cup S$, tels que $Q_i = V_{\rho(i)} \setminus \bigcup_{j=i+1}^{n} V_{\rho(j)}$, où V_i constitue l'ensemble de variables d'entrée et d'état dont dépend effectivement T_i. Par construction, ces sous-ensembles sont deux à deux disjoints et leur union engendre l'intégralité de l'ensemble $X \cup S$. Le calcul de IMG comprend n étapes de calcul des images partielles IMGP_i :

$\text{IMGP}_1 = \exists v \in \mathbb{B} : T_{\rho(1)} \,.\, \chi_E$, pour chaque $v \in Q_1$
$\text{IMGP}_i = \exists v \in \mathbb{B} : T_{\rho(i)} \,.\, \text{IMGP}_{i-1}$, pour chaque $v \in Q_i$ et pour $i = 2..n$

Dans le cas particulier où $Q_i = \emptyset$, l'expression de l'image partielle devient :

$\text{IMGP}_i = T_{\rho(i)} \,.\, \text{IMGP}_{i-1}$

L'efficacité de cette approche dépend des propriétés de localité du modèle analysé, pour lequel il s'agit de résoudre le problème d'optimisation suivant : construire une permutation ρ permettant de maximiser la taille de chaque Q_i, tout en veillant à ce que le nombre de cas où $Q_i = \emptyset$ soit le plus petit possible. En d'autres termes, entre deux conjonctions il faut effectuer le plus de quantifications existentielles possibles.

Exemple

Calcul d'image pour le modèle ARB
On constate que chacun des composants T_1 et T_2 de la relation de transition du modèle ARB dépend de l'intégralité des variables $X \cup S$. Dans cette situation, quelle que soit la permutation ρ, on a $Q_1 = \emptyset$ et $Q_2 = X \cup S$. Dans ce cas, les images partielles prennent la forme :

$\text{IMG}\text{P}_1 = T_{\rho(1)} \cdot \chi_0$,
$\text{IMG}\text{P}_2 = \exists v \in \mathbb{B} : T_{\rho(2)} \cdot T_{\rho(1)} \cdot \chi_0$, pour chaque $v \in X \cup S$
Ce scénario est défavorable, car il nécessite le calcul de la relation de transition $T_1 \cdot T_2$.

Calcul d'image pour le modèle ARB'
La dépendance des composants T_1, T_2 et T_3 par rapport aux variables d'entrée et d'état est la suivante :

- T_1 dépend des variables x_1, x_2, x_3, s_1 ;
- T_2 dépend des variables x_1, x_2, x_3, s_1, s_2 ;
- T_3 dépend des variables $x_1, x_2, x_3, s_1, s_2, s_3$.

La permutation $\rho = (1, 2, 3) \rightarrow (3, 2, 1)$ permet de construire les sous-ensembles $Q_1 = \{s_3\}$, $Q_2 = \{s_2\}$ et $Q_3 = \{x_1, x_2, x_3, s_1\}$ et on obtient les images partielles :
$\text{IMG}\text{P}_1 = \exists s_3 \in \mathbb{B} : T_3 \cdot \chi_0$,
$\text{IMG}\text{P}_2 = \exists s_2 \in \mathbb{B} : T_2 \cdot \text{IMG}\text{P}_1$,
$\text{IMG}\text{P}_3 = \exists x_1, x_2, x_3, s_1 \in \mathbb{B} : T_1 \cdot \text{IMG}\text{P}_2$.
Ainsi, ce scénario permet de calculer l'image de façon progressive, en utilisant séparément les composants de la relation de transition. Il a donc été possible d'éviter la construction de la relation de transition globale.
□

La partition de la relation de transition. Cette approche exploite une propriété de localité de nature différente : les composants T_i peuvent être regroupés dans r partitions $P_1, ..., P_r$, de taille $k_1, k_2, ...k_r$, telles que leurs sous-ensembles support sont deux à deux disjoints. Encore une fois, il s'agit de trouver une permutation ρ sur les composants de T, permettant de construire les partitions :

$P_1 = \prod_{j=1}^{K_1} T_{\rho(j)}$
$P_i = \prod_{j=K_{i-1}+1}^{K_i} T_{\rho(j)}$

où $K_i = \sum_{p=1}^{i} k_p$. Ainsi, l'expression de l'image peut s'écrire :

$$\chi_{\text{IMG}(T,\chi_E)} = \exists \langle x_m \rangle \in \mathbb{B}^m, \exists \langle s_n \rangle \in \mathbb{B}^n : \prod_{i=1}^{r} P_i \cdot \chi_E \tag{11}$$

Dans ce contexte, c'est la structure de la fonction caractéristique χ_E qui détermine à partir de quel niveau il est possible de séparer les quantifications des partitions P_i. Si χ_E est un produit de littéraux appartenant à l'ensemble S, alors il peut lui même être éclaté en r partitions $\chi_{E_1}, ..., \chi_{E_r}$ dont les supports sont deux à deux disjoints. La partition est construite de façon à ce que le support de χ_{E_i} soit inclus dans le support de P_i. Dans ce cas, l'expression de l'image peut se réécrire :

$$\chi_{\text{IMG}(T,\chi_E)} = \prod_{i=1}^{r} \left(\exists \langle x_m \rangle \in \mathbb{B}^m, \exists \langle s_n \rangle \in \mathbb{B}^n : P_i \cdot \chi_{E_i} \right) \tag{12}$$

En revanche, si l'expression de χ_E est plus complexe, alors ce genre de décomposition permettra uniquement de distribuer la quantification existentielle des variables d'entrée sur les différentes partitions P_i. Ceci peut entraîner des simplifications importantes si chacune

de ces partitions dépend d'un grand nombre de variables d'entrée. Il suffit d'ailleurs d'effectuer cette quantification une seule fois, au tout début du parcours symbolique du modèle. Les partitions résultantes dépendent uniquement de variables d'état et peuvent ensuite être recombinées grâce à la technique de quantification "au plus tôt". Cette technique sera illustrée au paragraphe §2.4.3.4

1.4.4.2 La méthode fonctionnelle

Cette méthode utilise la relation de transition du modèle stockée comme un vecteur de n fonctions de transition[69], d'où son nom. Elle emploie la décomposition de Boole appliquée à la simplification d'une fonction (cf. §1.4.3). Combinée avec la quantification existentielle, la décomposition de Boole permet de décomposer le calcul d'image en sous-problèmes qui ne dépendent plus des variables quantifiées. Soit $\langle v_{m+n}\rangle = \langle x_m\rangle \bullet \langle s_n\rangle$ le vecteur obtenu par concaténation des variables d'entrée et d'état de M. Soit $v_j, 1 \leq j \leq m+n$ un de ses composants.

Par application de la décomposition selon v_j l'équation 6 de l'image peut se réécrire :

$$\exists v_1, ..., v_j, ...v_{m+n} : v_j \cdot \prod_{i=1}^{n} \left(s'_i \Leftrightarrow F_t^i(\langle v_{m+n}\rangle)\right)\big|_{v_j} \cdot \chi_E(v_{m+1}...v_{m+n})\big|_{v_j} +$$

$$\exists v_1, ..., v_j, ...v_{m+n} : \bar{v}_j \cdot \prod_{i=1}^{n} \left(s'_i \Leftrightarrow F_t^i(\langle v_{m+n}\rangle)\right)\big|_{\bar{v}_j} \cdot \chi_E(v_{m+1}, ..., v_{m+n})\big|_{\bar{v}_j} \qquad (13)$$

Après quantification existentielle de la variable v_j on obtient :

$$\chi_{\text{IMG}(T,\chi_E)} = \exists v_1, ..., v_{j-1}, v_{j+1}, ...v_{m+n} : \prod_{i=1}^{n} \left(s'_i \Leftrightarrow F_t^i(\langle v_{m+n}\rangle)\right)\big|_{v_j} \cdot \chi_E(v_{m+1}, ..., v_{m+n})\big|_{v_j} +$$

$$\exists v_1, ..., v_{j-1}, v_{j+1}...v_{m+n} : \prod_{i=1}^{n} \left(s'_i \Leftrightarrow F_t^i(\langle v_{m+n}\rangle)\right)\big|_{\bar{v}_j} \cdot \chi_E(v_{m+1}...v_{m+n})\big|_{\bar{v}_j} \qquad (14)$$

En alternant décomposition de Boole et quantification existentielle, l'expression de χ_{IMG} se transforme en une somme de 2^{m+n} produits impliquant chacun n littéraux parmi $s'_1, ...s'_n$. En réalité, il suffit de calculer l'ensemble de cofacteurs $F_t^i\big|_{C_j}, i = 1..n, j = 1..2^{m+n}$ tels que $\chi_E\big|_{C_j} = 1$. L'expression de l'image devient :

$$\chi_{\text{IMG}(T,\chi_E)} = \sum_{j=1}^{2^{m+n}} \prod_{i=1}^{n} s'_i \cdot F_t^i(\langle v_{m+n}\rangle)\big|_{C_j} \qquad (15)$$

La méthode fonctionnelle apporte une amélioration considérable en ce qui concerne la taille des BDDs manipulés. La relation de transition est stockée en tant que vecteur de fonctions de transition, à raison d'un BDD par fonction. En pratique, chacun de ces BDDs a une taille relativement réduite, de l'ordre de la centaine, voire un millier de nœuds. Ce stockage est largement moins coûteux que la construction de la relation de transition monolithique. Par ailleurs, contrairement à l'approche relationnelle, il n'y a pas d'explosion combinatoire *pendant* le calcul de l'image. Le seul BDD intermédiaire qui est manipulé est celui contenant le résultat et sa construction se fait par calcul successif de $|E|$ cofacteurs, où E est l'ensemble d'états représenté par χ_E.

Remarques

1. En pratique, calculer les cofacteurs de F_t^i sans annuler l'expression de χ_E revient à restreindre les domaines de définition des F_t^i. L'opérateur de contrainte $\downarrow$ [26] permet d'exprimer cette restriction : $F_t^i \downarrow \chi_E$ dénote la restriction du domaine $\mathbb{B}^{m+n}$ de F_t^i aux valeurs pour lesquelles $\chi_E = 1$.
2. Les variables de prochain état s_i' n'interviennent pas dans le calcul de l'image, mais seulement dans la construction du résultat.

Il est clair que l'approche fonctionnelle présente un grand intérêt dans le cas où le cardinal de l'ensemble de départ $|E|$ reste polynomial en n. En revanche, si $\chi_E = 1$ alors sa complexité reste exponentielle en nombre de pas de calcul. Ainsi, son application aura le plus d'efficacité en début du parcours symbolique : l'ensemble initial contient souvent un seul état. Si tel est le cas, alors le premier calcul d'image coûtera $n2^m$ calculs de cofacteurs, 2^m pour chaque fonction de transition. Si par contre $\chi_E = 1$, alors $|E| = 2^n$ et le calcul d'image nécessitera $n2^{m+n}$ calculs de cofacteurs. Par ailleurs, on observe que le nombre m de variables d'entrée du modèle influence lui aussi la complexité du calcul d'image, limitant l'application de cette technique aux modèles ayant un nombre réduit de variables d'entrée.

1.4.5 Tour d'horizon des techniques de parcours symbolique

La *simplification de l'ensemble de frontière* [69] est une technique appliquée au calcul d'ensemble d'états atteignables. Ce calcul comporte plusieurs itérations de point fixe. A chaque itération, comme le montre l'équation (8), on calcule l'image de l'ensemble d'états déjà atteints S_i à travers la relation de transition T. L'ensemble d'états nouvellement atteints est S_{i+1}. On appelle ensembles *frontière* les ensembles $FR_{i+1} = S_{i+1} \setminus S_i$, avec $FR_0 = S_0$. A tout moment, on a : $FR_i \subseteq S_i$. L'application successive du calcul d'image sur les ensembles frontière FR_i obtenus lors de chaque itération permet également de parcourir l'intégralité de l'espace d'états du modèle. Il est donc intéressant d'employer les ensembles FR_i de cardinalité plus petite que celle de S_i, et dont la représentation par BDD est potentiellement plus simple. Un algorithme est proposé pour construire une approximation des ensembles de frontière FR_i ayant une représentation BDD plus simple : en ajoutant judicieusement des états dans l'ensemble FR_i, l'approximation FR_i' obtenue peut avoir une représentation beaucoup plus simple. Le résultat obtenu vérifie la propriété : $FR_i \subseteq FR_i' \subseteq S_i$.

Contrairement aux approches relationnelle et fonctionnelle, cette technique s'attaque à la complexité du terme χ_E intervenant dans le calcul d'image. Son utilisation est intéressante dans le cas précis du calcul des états atteignables d'un modèle, mais devient problématique dans toute autre application à cause du caractère approximatif des ensembles d'états manipulés. L'idée d'approximation d'ensembles d'états a été récemment utilisée pour la vérification de modèles appliquée à une classe restreinte de propriétés [72].

Le *parcours logarithmique*[8] [54] est une méthode de calcul de l'espace d'états atteignables qui a pour but de réduire le nombre d'itérations de point fixe nécessaires au parcours de l'espace d'états d'un modèle. Le principe de cette méthode est de calculer

[8] en anglais : *iterative squaring*

d'abord T^*, la fermeture transitive de la relation de transition T. La fermeture transitive est elle-même le point fixe de l'équation récurrente :

$$T^{2^{i+1}}(\langle x_m\rangle, \langle s_n\rangle, \langle s'_n\rangle) =$$
$$T(\langle x_m\rangle, \langle s_n\rangle, \langle s'_n\rangle) + \exists \langle x_m\rangle, \langle z_n\rangle : T^{2^i}(\langle x_m\rangle, \langle s_n\rangle, \langle z_n\rangle) \,.\, T^{2^i}(\langle x_m\rangle, \langle z_n\rangle, \langle s'_n\rangle)$$

où $T^1 = T$. Conformément, à cette équation, les états s_n et s'_n font partie du résultat s'il existe au moins un chemin partant de s_n et arrivant à s'_n. Le nombre d'itérations nécessaire à l'obtention du résultat est le logarithme de la plus grande distance entre deux états du modèle parcouru. L'ensemble d'états atteignables est donné par l'image de l'état initial à travers la fermeture transitive de T : $\chi_{\mathrm{IMG}(T^*,\chi_0)}$.

Cette technique présente de l'intérêt lorsque le parcours symbolique classique nécessite un grand nombre de pas : ses performances sont incontestables dans le cadre du parcours symbolique des compteurs. Cependant, appliquée de façon systématique, elle souffre d'explosion combinatoire lors des étapes intermédiaires de calcul des T^{2^i}. Dans [16] les auteurs proposent d'utiliser de manière sélective le parcours logarithmique, sur des modèles séquentiels contenant aussi des compteurs. La relation de transition est décomposée de façon disjonctive. Cette décomposition a pour but de scinder le graphe d'état du modèle en parties disjointes. Chaque composant de la relation de transition caractérise une unique partie dans le graphe d'état. Pour obtenir l'ensemble d'états du modèle, on calcule l'ensemble des états atteignables pour chaque composant de la relation de transition, en utilisant un algorithme adéquat. Les composants (connus par l'utilisateur) ayant une profondeur de parcours élevée seront parcourus avec la technique logarithmique.

Le *parcours mixte* [66] implémente le choix *dynamique* entre la méthode relationnelle ou fonctionnelle dans le calcul de l'image ou de la pré-image. Pour chaque pas du parcours symbolique, on décide si la méthode la plus appropriée est la quantification au plus tôt, la partition de la relation de transition, ou bien le calcul fonctionnel. Cette décision s'appuie sur les qualités structurelles du modèle : une *matrice de dépendance* $D(n+1, m+n)$ exprime de manière synthétique cette structure : $D(i,j) = 1$ si la fonction de transition F_t^i dépend de la variable $v_j \in X \cup S$ et 0 sinon. La $n+1$-ième ligne dans D définit les mêmes dépendances pour la fonction caractéristique χ_E de l'ensemble d'états courants.

La plupart des modèles rencontrés dans la pratique ont une propriété de localité : les supports des fonctions de transition constituent chacun une proportion négligeable de l'ensemble $X \cup S$. Cette propriété se traduit par une matrice de dépendance D *creuse* et suggère l'emploi de l'approche relationnelle pour le calcul d'image (cf. §1.4.4.1) :

- si les lignes et les colonnes de D peuvent être permutées de façon à obtenir une nouvelle matrice creuse à distribution bloc-diagonale, alors les fonctions de transition du modèle ont des supports disjoints. La relation de transition peut être partitionnée ;
- si D peut être retranscrite sous forme bloc-triangulaire, alors il existe un ordonnancement efficace de variables permettant d'appliquer la quantification au plus tôt.

Les auteurs montrent que dans la plupart des cas, l'approche relationnelle reste plus efficace que l'approche fonctionnelle. Ils cherchent à appliquer le parcours relationnel. En revanche, si la matrice de dépendance du modèle est *dense*, alors la relation de transition ne peut pas être décomposée. Le calcul d'image utiliserait une relation de transition monolithique, ce qui est à éviter. Dans cette situation, la méthode fonctionnelle est utilisée pour obtenir une décomposition disjonctive récursive de la relation de transition. Si le niveau de récursion

est k, le calcul d'image est décomposé en 2^k sous-buts. Une matrice de dépendance est construite pour chacun de ces sous-buts, et son contenu est de nouveau analysé. Après la décomposition disjonctive, au moins k variables on été éliminées. Chacune de ces matrices est donc forcément moins dense que la matrice du modèle initial. Le choix des variables pour la décomposition est basé sur leur fréquence d'apparition dans les supports des fonctions de transition ou des partitions.

Si, après décomposition disjonctive, les tailles des BDDs n'ont pas décru de manière significative, ou si les matrices de dépendance obtenues ne deviennent pas creuses, alors le parcours relationnel prend le relais.

Cette approche hybride constitue un pas important en direction de la robustesse du parcours symbolique. Dans un nombre important de cas, ses performances dépassent celles des parcours relationnel ou fonctionnel employés séparément.

1.4.6 Conclusion

Dans la pratique, le parcours symbolique des modèles synchrones se heurte très souvent aux limites exponentielles de complexité, que ce soit en taille de représentation ou en ressources de calcul. Un des principaux facteurs conduisant à de tels problèmes de complexité est la forme conjonctive de la relation de transition, rendant a priori impossible la décomposition du calcul d'image ou de pré-image. En dehors des cas où le modèle possède certaines propriétés de localité, la relation de transition ne peut pas être préalablement décomposée avant le parcours symbolique. Le dernier recours dans une telle situation reste la décomposition de Boole. Le choix des variables pour la décomposition se base sur leur fréquence d'apparition dans les composants d'une relation de transition.

Les techniques SAT deviennent de plus en plus performantes. Dans [25], l'approche de vérification de systèmes bornés est appliquée dans un contexte industriel. Deux outils sont comparés : l'un utilise SAT et l'autre les BDDs. Compte-tenu de l'état de l'art dans la résolution SAT, cette approche est plus robuste que celle basée sur les BDDs. Cette dernière nécessite du guidage supplémentaire de la part de l'utilisateur, notamment en ce qui concerne l'ordre des variables.

Chapitre 2

Spécification : système et environnement

2.1 Introduction

Ce chapitre est consacré aux méthodes de spécification des systèmes séquentiels modélisés par une machine d'états finis (FSM). Dans ces systèmes, toute transition correspond au passage d'une unité de temps, indiqué par les événements discrets d'un mécanisme d'horloge. Dans la suite, on identifie sans perte de généralité le système résultant à son modèle FSM sous-jacent.

Le processus de réalisation d'un système comporte nécessairement une étape d'analyse et une étape ultérieure de synthèse. Le but de la démarche analytique est de raffiner l'ensemble des besoins auxquels le système doit répondre. Pour chaque besoin on décrit un traitement y répondant au mieux. Quant à la démarche synthétique, son but est de répondre à la question "le système obtenu satisfait-il les besoins initiaux" ? En occurrence, si $P_1, ..., P_N$ sont des propriétés - prédicats logiques - exprimant ces besoins, on souhaite s'assurer que le modèle FSM M sous-jacent au système résultant satisfait chacune des propriétés P_i. Bien souvent, cette question n'a qu'une réponse très partielle, donnée par la simulation classique : M peut être amené à satisfaire P_i, pour un vecteur de test adéquat. L'ensemble $P_1, ..., P_N$ forme une spécification pour le modèle M. Dans la suite de ce document, on utilise la notation $M \models P$ pour exprimer le fait que le modèle M satisfait la propriété P. De même, on note $e \models_M P$ le fait que la propriété P soit vraie dans l'état e du modèle M. En absence d'ambiguité, l'indice de cet opérateur sera omis.

Il existe plusieurs sortes des spécifications. Soit P un prédicat logique exprimé sur des variables de $X \cup S \cup O$. L'affirmation "M satisfait P" possède plusieurs interprétations. Si P doit être satisfait dans tous les états atteignables de M, alors P est dit un *invariant* dans M. Afin de pouvoir raisonner sur le comportement séquentiel de la plupart des systèmes modélisés aujourd'hui, différentes *logiques temporelles* permettent de combiner des opérateurs temporels dans le but d'exprimer, outre l'invariance, la causalité et la succession. En particulier, les formules temporelles peuvent exprimer des invariants. Si P est une formule temporelle alors $M \models P$ si et seulement si P est vraie dans l'état initial de M. Un ensemble de formules temporelles constitue une *spécification logique* de M.

Le fait de rassembler la totalité des formules temporelles constituant une spécification logique peut s'avérer difficile, à mesure que croît la complexité du système spécifié. Certains

parmi ces prédicats peuvent être rendus implicites, à travers une description opérationnelle codée sous la forme d'une machine d'états. Si P est une formule temporelle et M_P une machine d'états qui la satisfait, alors prouver que $M \models P$ revient à établir un lien entre les modèles M et M_P : en pratique cela se traduit par une preuve de simulation (raffinement) ou d'équivalence séquentielle. Le modèle M_P constitue une *spécification opérationnelle* de M.

Pour des raisons de complexité des systèmes modélisés, il est rarement possible d'établir formellement si M satisfait ou non sa spécification $P_1, ...P_N$. En pratique, pour cette tâche, la *décomposition* se montre indispensable à deux niveaux :

- si possible décomposer chacune des propriétés P_i en sous-éléments. Ceux-ci peuvent être *explicites* ou *implicites*. Par exemple, soit $P_i = A.B$, où A exprime une condition d'exclusion mutuelle et B exprime une condition de correction d'une donnée. A et B sont des sous-éléments explicites de P_i. Prouver P_i revient à prouver la validité de A et de B séparément. En revanche, la mise en œuvre de l'exclusion mutuelle A peut faire le choix entre deux algorithmes d'arbitrage différents, A_1 et A_2. Prouver A, revient à prouver que les deux algorithmes d'arbitrage sont corrects. A_1 et A_2 sont donc des éléments implicites de P_i ;
- si M est un modèle hiérarchique, associer autant que possible les éléments de P_i avec des composants de M. Par exemple, il suffit d'évaluer A_1 à l'échelle locale du composant de M qui met en œuvre l'algorithme d'arbitrage.

Une spécification est donc un "contrat", que l'implémentation doit respecter lorsqu'elle est placée dans un environnement approprié. La nature de l'environnement constitue une véritable contrainte pour l'évaluation d'une spécification. Il est donc souvent nécessaire d'exprimer des prédicats limitant les valeurs ou les séquences de valeurs possibles sur les variables d'entrée $\langle x_n \rangle$; en effet, il est rare qu'un système fasse preuve de robustesse, au point de pouvoir satisfaire l'ensemble de ses spécifications quel que soit son environnement.

Ce chapitre passe en revue les méthodes de spécification d'un circuit numérique et de son environnement, ainsi que les algorithmes symboliques de preuve associés. L'association d'une méthode de spécification à un algorithme de preuve est ensuite discutée dans le cadre d'une démarche préliminaire de vérification axée sur la gestion efficace de l'explosion combinatoire.

2.2 Spécifications logiques - la logique temporelle

La logique temporelle permet d'exprimer des prédicats - formules temporelles - sur des états individuels ou sur des séquences d'états d'un modèle. Compte tenu du caractère implicite du passage du temps, la sémantique temporelle de ces prédicats est restreinte à la succession et à la causalité. Soient P et Q deux prédicats satisfaisables dans le modèle M :

- Q est causé par P si et seulement si quel que soit l'état p de M tel que $p \models P$, toute séquence d'états commençant par p contient au moins un état q tel que $q \models Q$. Cependant, le fait qu'un état satisfasse le prédicat Q n'a aucune implication sur la satisfaisabilité de P dans M ;

– Q succède à P si et seulement si quel que soit l'état q de M tel que $q \models Q$, toute séquence d'états de M menant à q contient au moins un état p tel que $p \models P$. Cependant, le fait qu'un état satisfasse le prédicat P n'a aucune implication sur la satisfaisabilité de Q dans M.

Ces notions permettent d'orienter le raisonnement sur M soit vers le futur, pour la causalité, soit vers le passé, pour la succession. Elles expriment la conséquence ou l'antécédent à caractère certain. Il est toutefois possible de leur associer des notions correspondantes exprimant la possibilité au lieu de la certitude.

Deux interprétations sont possibles pour les définitions ci-dessus : linéaire (à partir d'un état, une seule transition est possible) et arborescente (à partir d'un état, plusieurs transitions sont possibles).

La première logique temporelle utilisable dans un contexte automatisé est la logique arborescente CTL[1] [22] proposée par Emerson et Clarke. Les mêmes auteurs définissent l'extension CTL* [23], permettant d'exprimer aussi bien des comportements linéaires qu'arborescents. Cette logique s'est montrée particulièrement adaptée à la spécification des circuits numériques.

2.2.1 La logique temporelle CTL*

Ce formalisme permet d'exprimer des formules temporelles *de chemin* ou *d'état* :

– les *formules de chemin* sont des prédicats rattachés à un ensemble de chemins (séquence d'états) du modèle. Un chemin d'états $\mathbf{e} = (e_0, e_1, ...), e_i \in \mathbb{B}^n$ d'un modèle M peut satisfaire ($\models$) les formules suivantes :

  1. tout prédicat P exprimé sur $X \cup S \cup O$. Le chemin $\mathbf{e}$ satisfait P, $\mathbf{e} \models P$, si et seulement si le prédicat P est vrai dans l'état e_0.
  2. $\mathbf{e} \models Xf$ si et seulement si $e_1 \models f$; f dénote un prédicat, une formule de chemin ou d'état ;
  3. $\mathbf{e} \models Ff$ si et seulement si $\exists i \geq 0$ tel que $e_i \models f$. f dénote un prédicat, une formule de chemin ou d'état ;
  4. $\mathbf{e} \models Gf$ si et seulement si $\forall i \geq 0 : e_i \models f$. f dénote un prédicat, une formule de chemin ou d'état ;
  5. $\mathbf{e} \models f_1 U f_2$ si et seulement si $\exists k \geq 0$ tel que $e_k \models f_2$ et $\forall i : 0 \leq i < k : e_i \models f_1$. $f_1 et f_2$ dénotent des prédicats, des formules de chemin ou d'état ;

– les *formules d'état* sont des prédicats rattachés à un ensemble d'états du modèle. Tout état $e \in \mathbb{B}^n$ de M se trouve en tête d'au moins un chemin $\mathbf{e} = (e, e_1, ...,)$, et peut satisfaire les formules suivantes :

  1. tout prédicat P exprimé sur $X \cup S \cup O$;
  2. $e \models Af$ si et seulement si pour tout chemin $\mathbf{e} = (e, ...,)$ commençant par e on a $\mathbf{e} \models f$;
  3. $e \models Ef$ si et seulement s'il existe un chemin $\mathbf{e} = (e, ...,)$ commençant par e tel que $\mathbf{e} \models f$.

[1] abrégé en anglais : *Computation Tree Logic*

CTL	LTL
AGf	Gf
AXf	Xf
AFf	Ff
$A(fUd)$	fUd
$AG(f \rightarrow AGd)$	$G(f \rightarrow Gd)$
$AG(f \rightarrow AFd)$	$G(f \rightarrow Fd)$
$AGAFf$	GFf
$AG(f \rightarrow A(dUh))$	$G(f \rightarrow (dUh))$

TAB. 2.1 – Formules CTL et LTL équivalentes

La logique CTL* constitue un cadre généralisé pour raisonner aussi bien sur des *arbres* que sur des *traces* d'exécution d'un modèle. Ses sous-ensembles "naturels" sont donc les logiques CTL et LTL.

La logique CTL est une restriction de CTL* qui permet de raisonner sur des arbres d'exécution. Dans une formule CTL bien formée tout opérateur parmi X, F, G, U est obligatoirement précédé par un quantificateur de chemin. Quant à LTL, elle ne permet de raisonner que sur un unique chemin d'exécution à la fois. Toute formule LTL valide est de la forme Af, où f est une formule de chemin. Les formules d'état sont exclues de f à l'exception des prédicats P exprimés sur $X \cup S \cup O$. En d'autres termes, les seuls opérateurs pouvant apparaître dans f sont X, F, G et U.

Dans la pratique, les outils de vérification supportent soit CTL, soit LTL, soit des variantes de ces logiques comme il sera montré dans la suite de ce chapitre. Ces deux logiques ont été départagées en raison de leur complexité. La preuve d'une formule CTL est linéaire en taille de la formule et du graphe d'état du système analysé [19]. En revanche, la preuve d'une formule LTL est exponentielle en taille de la formule à prouver même si elle reste linéaire en taille du graphe d'état.

Sur le plan expressif, la logique linéaire offre un cadre plus "naturel" pour la spécification que la logique arborescente. Il est toutefois possible de construire une relation d'équivalence sur CTL*, à partir de sous-ensembles stricts de CTL et LTL : une formule CTL P_{CTL} est équivalente à une formule LTL P_{LTL} si et seulement si quel que soit le modèle M, $M \models P_{CTL} \Leftrightarrow M \models P_{LTL}$. Le tableau 2.1 montre quelques formules CTL usuelles et leur correspondants LTL équivalents. En dehors de cette relation d'équivalence, les formules CTL et LTL ne sont pas comparables.

Remarque. Parmi les formules CTL* il existe deux classes de propriétés d'une grande importance pratique :

- *la sûreté* : si la formule P est fausse, il existe un contre-exemple (une séquence d'états du modèle) de longueur finie infirmant P ;
- *la vivacité* : si la formule P est fausse, il est impossible de construire un contre-exemple de taille finie.

En termes naturels, la sûreté exprime le fait que "quelque chose de mauvais ne se produit jamais", alors que la vivacité exprime le fait que "quelque chose d'attendu finira par arriver".

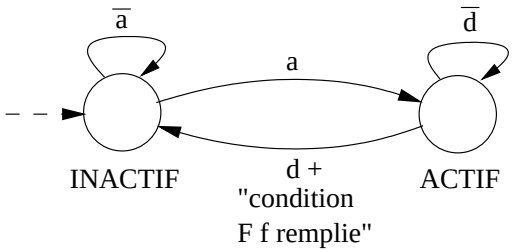

FIG. 2.1 – Automate de phase

2.2.2 Logiques temporelles à utilisation industrielle

2.2.2.1 Le sous-ensemble ∀-CTL

Connu également sous le nom ACTL, cette restriction de CTL bannit l'utilisation de la quantification existentielle sur des chemins. Le seul quantificateur autorisé est A. En outre, toute formule ∀-CTL doit respecter la *forme normale positive* : les opérateurs de négation s'appliquent exclusivement aux propositions atomiques. Ce sous-ensemble possède deux propriétés importantes :

1. Conservativité vis-à-vis d'une démarche compositionnelle de vérification. Soit C un modèle et $P_{\forall-CTL}$ une propriété telle que $C \models P_{\forall-CTL}$. Si C est réutilisé en tant que composant d'une machine hiérarchique de Mealy M, alors $M \models P_{\forall-CTL}$. Ainsi, toute formule ∀-CTL vraie à l'échelle réduite d'un composant du modèle M, reste vraie à l'échelle globale de M ;
2. Conservativité vis-à-vis de l'abstraction. Si M_{abs} constitue une spécification opérationnelle abstraite (et éventuellement non-déterministe) d'un modèle M_{impl}, telle qu'il existe une relation de simulation entre M_{abs} et M_{impl} alors toute formule ∀-CTL satisfaite par M_{abs} est également satisfaite par M_{impl} [46].

2.2.2.2 La logique linéaire LTL$^+$ et le macro-langage FQL

Utilisée actuellement au sein de l'outil de vérification FormalCheck de Cadence, la logique temporelle LTL$^+$ est une extension de la logique LTL permettant d'exprimer *la phase*. En plus des propositions atomiques, une formule LTL$^+$ peut référencer les états - ACTIF ou INACTIF[2] - d'un automate auxiliaire appelé automate de phase (Figure 2.1). Les transitions sont étiquetées par les prédicats a-condition d'activation et d-condition de désactivation, exprimés sur l'ensemble des variables du modèle ($X \cup S \cup O$). Les formules LTL$^+$ expriment des comportements qui doivent avoir lieu lorsque l'automate de phase se trouve dans l'état ACTIF. En d'autres termes, ces comportements doivent se produire à l'intérieur d'une "fenêtre temporelle de vérification", dont le début est marqué par a et la fin est marquée par d. Les comportements qui peuvent être exprimés au sein d'une telle fenêtre, sont l'invariance (G), l'inévitabilité (F), la répétition (GF) et l'invariance inévitable (FG).

[2]de l'anglais : *ENABLED* et *DISCHARGED*

$a = 1$ $d = 0$	Gf, Ff, FGf
$a = 1$ d quelconque	$Gf + fUd$, $Ff + Fd$, $FGf + Fd$
a quelconque $d = 0$	$G(a \rightarrow XGf)$, $G(a \rightarrow FGf)$ $G(GFa \rightarrow Ff)$ $G(a$. INACTIF $\rightarrow XFf)$
a quelconque d quelconque	$G(a$. INACTIF $\rightarrow X(Gf + fUd))$, $G(a$. INACTIF $\rightarrow X(Ff + Fd))$, $G(a$. INACTIF $\rightarrow X(FGf + Fd))$.
$a = 0$ ou $d = 1$	$G(1)$

TAB. 2.2 – Formules LTL$^+$ acceptées par l'outil FormalCheck

L'état initial de cet automate est INACTIF. Lorsque la preuve le requiert, cet automate est composé avec le modèle à vérifier.

La forme d'une formule LTL$^+$ acceptée par l'outil FormalCheck varie selon les valeurs des prédicats a et d :

- si $a = 1$ et $d = 0$ alors on considère que la largeur de la fenêtre de vérification est infinie ;
- si $a = 1$ et d est un prédicat quelconque, alors la fenêtre de vérification commence dès l'état initial et se referme lorsque d devient vrai ;
- si $d = 0$ et a est un prédicat quelconque, alors la largeur de la fenêtre de vérification est infinie, mais celle-ci ne commence que lorsque a devient vrai pour la première fois.
- si a et d sont des prédicats quelconques, la largeur de la fenêtre de vérification est définie par les transitions de l'automate de phase : le début est marqué par le passage à l'état ACTIF ; en revanche, la fin est marquée par l'instant où d devient vrai, instant qui précède le retour de l'automate à l'état INACTIF.
- les cas $a = 0$ ou $d = 1$, correspondent à des formules trivialement vraies.

Soit f un prédicat booléen exprimé sur les variables de $X \cup S \cup O$. L'ensemble des formules LTL$^+$ couramment acceptées par l'outil FormalCheck est donné dans le tableau 2.2.

En pratique, il est possible d'enrichir la spécification de la fenêtre de vérification en précisant un délai pour retarder son ouverture, et/ou un délai pour avancer sa fermeture.

On remarque que l'ensemble des formules LTL$^+$ présenté ci-dessus n'est pas défini de manière récursive. Il s'agit plutôt d'un sous-ensemble de formules temporelles linéaires ayant la particularité d'être faciles à écrire et à comprendre, et dont l'usage est "fréquent". Au sein de l'outil FormalCheck, ces formules s'expriment grâce à un langage de macros intitulé FQL[3][17]. Chaque formule temporelle contient deux clauses, **After** et **Unless**, permettant de spécifier la largeur de la fenêtre temporelle de preuve, et une clause exprimant l'obligation de preuve :

[3] abrégé en anglais : *FormalCheck Query Language*

After/IfRepeatedly :	a
Always/Eventually/EventuallyAlways :	f
Unless :	d

Toute formule temporelle citée par le tableau 2.2 possède un correspondant FQL unique. Les prédicats a et d sont ceux qui déterminent les transitions de l'automate de phase (figure 2.1). Ainsi, lorsque a devient vrai, le prédicat f doit être satisfait aussi longtemps que l'automate de phase reste dans l'état $ACTIF$, selon l'obligation de preuve qui a été spécifiée :

- **Always** : f doit être vrai tant que la machine de phase reste dans l'état $ACTIF$, c'est à dire jusqu'à ce que d devienne vrai ;
- **Eventually** : f doit devenir vrai pendant que l'automate de phase se trouve dans l'état $ACTIF$. Si f ne peut jamais être satisfait dans l'état $ACTIF$, alors la propriété échoue. L'automate de phase retourne à l'état $INACTIF$ dès que f ou d sont satisfaites.
- **EventuallyAlways** : f doit devenir vrai et le rester, alors que l'automate de phase reste dans l'état $ACTIF$. Cette formule n'est satisfaisable que si cet état est conservé à l'infini (à partir d'un certain moment, la valeur de d sera toujours 0).

La clause **IfRepeatedly** a ne peut se combiner qu'avec **Eventually** f. Elle exprime le fait que si a est satisfait infiniment souvent, alors f doit être satisfait au moins une fois.

L'ensemble des formules LTL$^+$ supportées par l'outil FormalCheck est relativement réduit. Cette restriction est principalement dûe à la complexité de preuve caractéristique des formules temporelles linéaires. Au sein d'un outil de vérification industriel, un tel choix est justifié par l'intention d'éviter au maximum les situations où la preuve échoue par manque de ressources.

2.2.2.3 La logique arborescente CTL et le macro-langage SUGAR

Le macro-langage SUGAR[4] est le langage de spécification de l'outil RuleBase d'IBM. Historiquement, il formait une sur-couche de la logique temporelle CTL, permettant d'exprimer des formules temporelles d'une manière à la fois concise et paramétrée. Il a servi de point de départ pour la définition d'un langage de spécification de propriétés en voie de standardisation par l'organisme IEEE. Il offre quatre principaux mécanismes de spécification :

1. la logique temporelle linéaire LTL. Contrairement à FQL, SUGAR autorise l'imbrication des opérateurs linéaires ;
2. la logique temporelle arborescente CTL ;
3. les expressions régulières étendues. Il s'agit de prédicats de la forme $seq = p_1[d_1]; p_2[d_2]; ...$ exprimant une séquence d'événements booléens p_i. Chaque événement peut être un prédicat booléen ou une expression régulière. Les prédicats p_i peuvent être accompagnés par une indication de durée d_i, exprimée en nombre d'états (consécutifs ou non), pendant lesquels p_i doit être vrai. En absence d'un indicateur de durée, l'obligation de validité de p_i se restreint à un seul état dans seq.

Certaines expressions régulières peuvent être exprimées grâce à CTL. Ainsi, les deux formules ci-dessous sont équivalentes :
$AG\ p; q; r \Leftrightarrow AG\ \big(p\ .\ AX\ (q\ .\ AX\ r)\big)$
Les durées d_i sont généralement des intervalles. Il est possible de spécifier si les occurrences de p_i doivent être consécutives - $d_i = [*i..j]$, $i \leq j$, ou non - $d_i = [= i..j]$, $i \leq j$. La spécification de durée consécutive peut elle aussi se réécrire en CTL :
$AG\ (p; q[*i..j]; r) \Leftrightarrow AG\ p\ .\ \big(\underbrace{AX\ (q\ .\ AX\ (q\ .\ AX(\ldots AX\ (q}_{i \text{ fois}}\ .\ AX\ r))+$
$\underbrace{AX\ (q\ .\ AX\ (q\ .\ AX(\ldots AX\ (q}_{i+1 \text{ fois}}\ .\ AX\ r)) + \ldots$
$\underbrace{AX\ (q\ .\ AX\ (q\ .\ AX(\ldots AX\ (q}_{j \text{ fois}}\ .\ AX\ r))\big)$
Pour le cas particulier où $i = j$ on obtient un nombre d'occurrences consécutives fixé $[*i]$. Il est également possible de spécifier des durées arbitraires, lorsque $i = 0$ et j n'est pas spécifié : $[*]$.
Au sein d'une expression régulière étendue, les séquences peuvent être combinées grâce aux opérateurs "et" et "ou".
Enfin, les opérateurs seq_1 |-> $seq2$ et seq_1 |=> seq_2 expriment la succession de deux séquences : si le comportement spécifié par seq_1 se produit, il sera nécessairement suivi par seq_2. De manière alternative, l'opérateur "->"exprime le fait que le début de seq_2 coïncide avec la fin de seq_1.
L'exemple ci-dessous illustre la différence entre les opérateurs "->" et "=>" :
$AG\ p_1; q_1; r_1$ |=> $p_2; q_2; r_2 \Leftrightarrow$

$$AG((p_1\ .\ AX\ (q_1\ .\ AXr_1)) \rightarrow AX\ AX\ AX(p_2\ .\ AX\ (q_2\ .\ AXr_2)))$$

alors que :
$AG\ p_1; q_1; r_1$ |-> $p_2; q_2; r_2 \Leftrightarrow$

$$AG((p_1\ .\ AX\ (q_1\ .\ AXr_1)) \rightarrow AX\ AX(p_2\ .\ AX\ (q_2\ .\ AXr_2)))$$

4. des opérateurs temporels complexes. Parmi ceux-ci, rappelons la définition d'une *fenêtre temporelle*, $within(p, q)(r)$: le prédicat r est vrai à partir de l'instant où p est vrai et jusqu'à l'instant exclus où q devient vrai. Des variations existent, pouvant exprimer l'inclusion ou l'exclusion des bornes p et q de la fenêtre temporelle où r doit être vrai. Cette formule reste vraie dans les cas où les bornes p ou q ne sont jamais atteintes.
   La *précédence*, $p\ before\ q$ exprime le fait que si le prédicat q est vrai, alors p doit être vrai ou bien il doit avoir été vrai ;
5. les instructions de macro-génération. Ce mécanisme permet à une formule temporelle de référencer un paramètre. Hormis certains cas particuliers basés sur la symétrie, une formule paramétrée nécessite d'être évaluée pour chaque combinaison possible de valeurs de ses paramètres.

Outre les expressions régulières, la particularité du langage SUGAR consiste en un éventail de macros qui étendent la notion "p est vrai dans le prochain état" à tout instant ultérieur

spécifié par le concepteur.

Comme il sera illustré dans la suite de ce chapitre, le langage SUGAR est étroitement lié à la notion de spécification opérationnelle. Ce lien implique des conséquences importantes sur l'efficacité de la preuve formelle. Il est également motivé par le besoin d'interopérabilité avec la simulation : la spécification opérationnelle correspondant à une formule temporelle est une machine d'états et peut de ce fait être simulée.

2.3 Spécifications opérationnelles

Une spécification opérationnelle est une machine d'états finis qui se voit attribuer le titre de "modèle de référence". Cette notion peut être utilisée dans différents contextes du flot de conception, avec à chaque fois une définition différente.

Dans le contexte d'une démarche de conception descendante, les caractéristiques principales d'une spécification opérationnelle sont l'abstraction et le non-déterminisme. L'implémentation est obtenue à travers une ou plusieurs étapes de *raffinement*. Une implémentation correcte ne doit pas s'écarter de sa spécification abstraite : tout comportement dans l'implémentation doit pouvoir être reproduit par la spécification.

Indépendamment de l'étape dans le flot de conception, une spécification opérationnelle peut être *réécrite*, sans aucun changement dans le niveau d'abstraction, que ce soit dans un but de conformité par rapport à un standard d'écriture, de maintenabilité ou d'optimisation. Contrairement au cas du raffinement, une réécriture correcte doit reproduire le même comportement que le modèle de référence.

Dans ces deux situations, il est difficile de distinguer objectivement la spécification opérationnelle de l'implémentation, les deux n'étant que des descriptions plus ou moins précises du même comportement.

Enfin, une spécification opérationnelle peut n'être qu'un simple "espion", n'englobant pas de connaissance sur le comportement du système, mais plutôt sur la nature de ses échanges avec l'environnement, et rapportant les éventuelles anomalies détectées. Une telle description est plus communément connue sous le nom de *moniteur*. Une implémentation est correcte si le moniteur ne peut rapporter aucune anomalie.

Il est important de noter que ces critères de correction n'ont de sens que si la spécification opérationnelle et son implémentation sont comparables (les noms des variables d'entrée, d'état et de sortie peuvent être mis en correspondance).

2.3.1 Modèles abstraits

La notion de modèle abstrait se confond avec celle de machine d'états finis. L'abstraction peut intervenir au niveau de la description des données manipulées ou bien des comportements à implémenter.

L'abstraction des données est synonyme de concision. Au sein d'une machine d'états finis $\mathcal{M}$, les symboles des alphabets d'entrée de sortie ou d'état sont plus abstraits (ou concis), par opposition aux symboles obtenus après l'encodage booléen de $\mathcal{M}$. Par exemple, l'alphabet des symboles qui représentent les états du modèle $\mathcal{ARB}$ (figure 1.1) contient quatre éléments. Pour des raisons pratiques, évoquées au chapitre précédent, l'encodage

booléen de $\mathcal{ARB}$ a engendré le modèle ARB'. Ce modèle contient trois variables booléennes d'état. Ainsi, un état est représenté par l'association de trois symboles, au lieu d'un seul. D'autre part, ces symboles peuvent encoder 8 états distincts.

Sur le plan comportemental, l'abstraction consiste à décrire *quel est* le comportement attendu, sans se préoccuper de *comment* celui-ci sera mis en œuvre dans l'implémentation. Le choix éventuel parmi plusieurs comportements peut être non-déterministe.

Plusieurs liens de conformité peuvent être établis entre une spécification opérationnelle abstraite et une implémentation, selon la sémantique, linéaire ou arborescente, sur laquelle on souhaite s'appuyer. Les approches linéaires (inclusion ou équivalence de traces) ne sont pas adéquates pour la preuve de circuits, n'étant pas suffisamment flexibles vis-à-vis de la composition parallèle en présence du non-déterminisme.

Les approches arborescentes de comparaison entre une spécification opérationnelle abstraite et une implémentation sont la simulation, le raffinement, la bisimulation et l'équivalence séquentielle.

La *simulation* se définit récursivement sur les états des deux modèles comparés (cf. §1.2.1, Définition 1.10). La preuve de simulation $\mathcal{M}_{impl} \preceq \mathcal{M}_{abs}$ a une complexité linéaire en nombre d'états de l'implémentation, mais exponentielle en nombre d'états de la spécification opérationnelle abstraite. Dans le domaine booléen symbolique, la définition de la simulation $M_{impl} \preceq_s M_{abs}$ est similaire à celle donnée pour le cas des modèles énumératifs. Soient $T^{impl}(\langle x_{m_1}\rangle, \langle s_{n_1}\rangle, \langle s'_{n_1}\rangle)$ la relation de transition de M_{impl} et et $T^{abs}(\langle y_{m_2}\rangle, \langle t_{n_2}\rangle, \langle t'_{n_2}\rangle)$ la relation de transition de M_{abs}. La relation symbolique de simulation $SIM(\langle s_{n_1}\rangle, \langle t_{n_2}\rangle)$ est le plus grand point fixe, solution de l'équation :

$$
\begin{aligned}
&Y(\langle s_{n_1}\rangle, \langle t_{n_2}\rangle) = \\
&\quad \forall \langle x_{m_1}\rangle, \forall \langle s'_{n_1}\rangle : \\
&\qquad T^{impl}(\langle x_{m_1}\rangle, \langle s_{n_1}\rangle, \langle s'_{n_1}\rangle) \quad \rightarrow \\
&\qquad \exists \langle y_{m_2}\rangle, \langle t'_{n_2}\rangle : T^{abs}(\langle y_{m_2}\rangle, \langle t_{n_2}\rangle, \langle t'_{n_2}\rangle) \ . \ Y(\langle s'_{n_1}\rangle, \langle t'_{n_2}\rangle)
\end{aligned} \tag{16}
$$

Pour déterminer si $M_{impl} \preceq_s M_{abs}$, il suffit de vérifier si les états initiaux $\langle s_0\rangle$ de M_{impl} et $\langle t_0\rangle$ de M_{abs} satisfont la relation $SIM(\langle s_0\rangle, \langle t_0\rangle)$.

Etant donné que M_{impl} et M_{abs} possèdent souvent des variables de sortie, il est possible d'ajouter dans l'équation ci-dessus une condition supplémentaire exprimant la nécessité que les fonctions de sortie des modèles comparés soient équivalentes.

Une variante restreinte de la notion de simulation, connue sous le nom de *raffinement* ($\preceq_r$), s'appuie sur la comparaison des fonctions de transition, plutôt que sur le dénombrement des états des deux modèles. Ainsi, lorsque la correspondance des noms des variables d'états est totale entre le modèle abstrait et l'implémentation, il est possible de construire une définition plus adéquate : toute transition qui a lieu dans l'implémentation peut être reproduite par le modèle abstrait. Compte-tenu du fait qu'un modèle abstrait est généralement plus compact que son implémentation, nous allons supposer sans perte de généralité que $X_{spec} \subseteq X_{impl}$.

Soient $m_2 \leq m_1$ deux entiers positifs. Soient $\langle x_{m_1}\rangle$ les variables d'entrée de M_{impl} et $\langle x_{m_2}\rangle$ les variables d'entrée de M_{abs}, telles que $\langle x_{m_1}\rangle = \langle x_1, \ldots x_{m_2}, x_{m_2+1}, \ldots x_{m_1}\rangle$. Soit $i \in \{1..n\}$ et $F^i_{impl}(\langle x_{m_1}\rangle, \langle s_n\rangle)$ une fonction de transition appartenant à M_{impl}. Soit la fonction F^i_{abs} le correspondant dans M_{abs} de F^i_{impl}. Dans le cas où M_{abs} est non-déterministe,

certaines parmi ses fonctions de transition ont la forme $F^i_{abs}(\langle x_{m_2}\rangle, \langle ND_r\rangle, \langle s_n\rangle)$, où $ND_j, j = 1..r$ sont des variables intitulées *pseudo-entrées*, de type booléen, servant à étiqueter les transitions non-déterministes de M_{abs}. Si, pour toute combinaison de valeurs $v_1, ...v_n \in \mathbb{B}^n$ appartenant à l'ensemble d'états atteignables de M_{impl}, on a :

$$\exists \langle ND_r\rangle : \forall i = 1..n : F^i_{impl}(\langle x_{m_1}\rangle, \langle v_n\rangle) \Leftrightarrow F^i_{abs}(\langle x_{m_2}\rangle, \langle ND_r\rangle, \langle v_n\rangle) \tag{17}$$

alors $M_{impl} \preceq_r M_{abs}$. En absence des pseudo-entrées, l'équation ci-dessus exprime la *bisimulation*.

La contrainte de correspondance totale entre les variables d'état des modèles comparés peut sembler toutefois très forte. La même relation de raffinement peut être conservée si toute variable d'état du modèle abstrait se retrouve également dans l'implémentation. La relation de raffinement se traduit ici par un invariant de transition. Soient T_{impl} et T_{abs} les relations de transition des modèles comparés. Soient $n_2 \leq n_1$ deux entiers positifs. Soient $\langle s_{n_1}\rangle$ les variables d'état de M_{impl} et $\langle s_{n_2}\rangle$ les variables d'état de M_{abs}, telles que $\langle s_{n_1}\rangle = \langle s_1, \ldots s_{n_2}, s_{n_2+1}, \ldots s_{n_1}\rangle$. Si, pour toute combinaison de valeurs $v_1, ...v_{n_1} \in \mathbb{B}^{n_1}$ appartenant à l'ensemble d'états atteignables de M_{impl}, on a :

$$T_{impl}(\langle x_{m_1}\rangle, \langle v_{n_1}\rangle, \langle v'_{n_1}\rangle) \rightarrow \exists \langle ND_r\rangle : T_{abs}(\langle x_{m_2}\rangle, \langle ND_r\rangle, \langle v_{n_2}\rangle, \langle v'_{n_2}\rangle) \tag{18}$$

alors $M_{impl} \preceq_r M_{abs}$.

Etant donné que la preuve de simulation est très coûteuse, car elle nécessite la manipulation de relations de transition monolithiques, nous nous focaliserons dans la suite sur sa version restreinte exprimant le raffinement.

2.3.2 Modèles de référence

La notion d'*équivalence séquentielle* est utilisée dans le cadre d'une approche "boîte noire", pour déterminer si les modèles comparés ont le même comportement *observable* (cf. §1.2.1, Définition 1.8). Les deux modèles booléens sont comparés en termes de fonctions de sortie. Contrairement au cas de la simulation, aucune correspondance entre les variables d'état n'est exigée. En revanche, les modèles comparés doivent avoir les mêmes variables d'entrée et de sortie. Soient M_{ref} un modèle booléen dit "de référence" et M_{impl} une implémentation des fonctionnalités de M_{ref}. Soit $i \in \{1..p\}$ et $F^i_{impl}(\langle x_m\rangle, \langle s_{n_1}\rangle)$ une fonction de sortie appartenant à M_{impl}. Soit la fonction $F^i_{ref}(\langle x_m\rangle, \langle s_{n_2}\rangle)$ le correspondant dans M_{ref} de F^i_{impl}. Si, pour toute combinaison de valeurs $v_1, ...v_{n_1+n_2} \in \mathbb{B}^{n_1+n_2}$ appartenant à l'ensemble d'états atteignables de la composition parallèle $M_{impl}||M_{ref}$, on a :

$$\forall i = 1..p : F^i_{impl}(\langle x_m\rangle, \langle v_{n_1}\rangle) \Leftrightarrow F^i_{ref}(\langle x_m\rangle, \langle v_{n_2}\rangle) \tag{19}$$

alors $M_{impl} \equiv M_{ref}$.

2.3.3 Moniteurs

Un moniteur est une spécification opérationnelle M_{spec} dont les fonctions de sortie expriment chacune une propriété de sûreté. Contrairement au cas d'un modèle abstrait ou

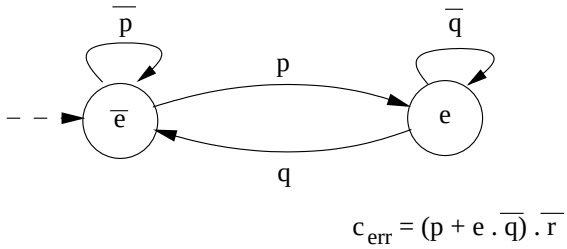

FIG. 2.2 – Moniteur pour la propriété SUGAR $within(p, q)(r)$

de référence, le moniteur *fait partie de l'implémentation*, étant systématiquement composé avec celle-ci, tout en restant passif vis-à-vis de son comportement. Pour cela, les conditions suivantes doivent être remplies :

- les ensembles de sortie du moniteur et de l'implémentation doivent être disjoints : $O_{spec} \cap O_{impl} = \emptyset$;
- le moniteur peut lire toutes les variables d'entrée et de sortie de l'implémentation : $X_{spec} \subseteq X_{impl} \cup O_{impl}$;
- le comportement du moniteur n'a pas d'influence sur l'implémentation : $O_{spec} \cap X_{impl} = \emptyset$;

Le graphe d'état de M_{spec} spécifiant la propriété de sûreté P se compose de deux composantes : une première composante S_P dont les états satisfont tous la propriété P et une composante S_{err} contenant un unique état dit "d'erreur". La relation de transition T_{spec} permet de passer de S_P à S_{err} lorsque P est violée, sans retour possible vers S_P. La transition depuis S_P vers S_{err} est étiquetée par un prédicat c_{err} intitulé *condition d'erreur*, exprimé sur les variables de $X_{spec} \cup S_{spec}$. La propriété P exprime le fait que la condition d'erreur n'est pas vraie et que l'état courant de M_{spec} n'est pas dans S_{err} :

$$P = \bar{c}_{err} \ . \ \langle s \rangle \notin S_{err}$$

Remarque. Compte tenu du style d'écriture du modèle M_{spec}, qui ne permet pas de retourner de S_{err} en S_P, la propriété P exprime en réalité deux aspects complémentaires :

1. $\bar{c}_{err}$: dans l'état courant, la condition d'erreur n'est pas vraie ;
2. $\langle s \rangle \notin S_{err}$: l'état courant de M_{spec} n'est pas dans S_{err} : la condition d'erreur n'a jamais été vraie.

Cette écriture présente surtout de l'intérêt dans le contexte de la réutilisabilité d'un moniteur dans un outil de simulation. Dans ce contexte, la présence de l'état puits S_{err} a le mérite de pouvoir "capturer" l'instant, au sein d'une longue trace de simulation, où P est devenue fausse. En revanche, pour la vérification formelle, le deuxième prédicat de P n'a aucune utilité, car cette propriété devient fausse dès l'instant où c_{err} devient vraie.

La propriété P est un invariant : elle doit être vraie dans tous les états atteignables du modèle composé $M_{impl}||M_{spec}$.

Une grande partie des propriétés temporelles écrites à l'aide des macros SUGAR v2.0 peuvent être directement traduites dans des moniteurs et prouvées comme des invariants.

Par exemple, la figure 2.2 représente un moniteur M_{within} permettant de spécifier la propriété SUGAR $within(p, q)(r)$. Le moniteur lit les variables p, q et r, et analyse la succession de leur valeurs : r doit être vrai entre l'instant où p devient vrai et jusqu'à l'instant précédant celui où q devient vrai. Cette propriété se traduit par l'assertion c_{err} suivante : "il se produit une erreur lorsque r est faux alors que p est vrai, ou lorsque r est faux pendant que M_{within} se trouve dans l'état étiqueté e et que q n'est pas vrai". La représentation de M_{within} fait abstraction de l'état puits. L'invariant à prouver est $P = AG\ \bar{c}_{err}$.

Un atout spécifique des moniteurs est la capacité de raisonner sur le passé. D'ailleurs, parmi les exemples ci-dessus, les formules qui référencent un futur borné sont reformulées en termes des événements du passé. Ce mécanisme est illustré au paragraphe §2.4.3.6.

2.3.4 Spécification de l'environnement

Une spécification d'environnement[4] relative à un modèle M exprime l'ensemble des valeurs admissibles pour les variables d'entrée $\langle x_n \rangle$ de M. Tout comme dans le cas d'un système, un environnement peut être décrit de manière logique ou opérationnelle.

La spécification logique d'un environnement est constituée d'un ensemble de prédicats pouvant exprimer plusieurs sortes de liens logiques :

- les valeurs des variables d'entrée peuvent être liées les unes par rapport aux autres ;
- les séquences de valeurs des variables d'entrée peuvent être influencées par les variables de sortie du système.

Selon les nécessités, ces liens peuvent être exprimés soit par des prédicats booléens, sous forme d'invariants, soit par des formules temporelles plus complexes. Lorsqu'il est inséré dans son environnement, le système fonctionne sous l'hypothèse que l'ensemble de ces prédicats sont vrais.

C'est la logique temporelle linéaire qui est la plus appropriée pour la spécification de séquences d'entrées acceptables. Ceci est justifié par une différence d'aspect temporel entre un système et son environnement. Dans le cas de la spécification d'un système, tout état peut posséder un ou plusieurs successeurs. A un instant donné et pour un état donné, le choix parmi les différents successeurs est déterminé par la valeur des variables d'entrée $\langle x_n \rangle$. Cet aspect confère à tout système à transitions une nature arborescente.

Par contraste, dans le contexte de la vérification, les valeurs des variables d'entrée d'un système peuvent se succéder de manière arbitraire, constituant ainsi un ensemble de séquences infinies (traces). Le fait de spécifier des contraintes d'environnement revient à définir des schémas de succession plus stricts, afin de filtrer les traces considérées comme inacceptables par le système. Il est donc plus naturel de raisonner de façon linéaire sur le comportement de l'environnement, en le visualisant comme un ensemble de traces.

L'expression de la vivacité lors de la spécification d'un système peut nécessiter l'introduction de *contraintes d'équité* au sein de son environnement. Soit p un prédicat logique utilisé dans la spécification d'une propriété de vivacité pour le système M. On peut avoir $s_0 \models AFp$ avec une sémantique arborescente, ou bien $s_0 \models Fp$ avec une sémantique linéaire. Deux cas de figure peuvent se présenter :

[4] notion connue également sous le nom de contrainte d'environnement

- de par la description de M, quelle que soit la séquence infinie d'entrées appliquées, il n'existe aucune séquence infinie σ d'états, telle que p soit faux dans chaque état de σ. La propriété de vivacité est satisfaite ;
- il existe au contraire une ou plusieurs séquences infinies d'entrées qui engendrent des séquences d'états de M le long desquelles p est toujours fausse. Si cette séquence traduit un comportement acceptable de l'environnement, alors la propriété de vivacité est fausse. Sinon, ce comportement doit être interdit.

Soit c un prédicat booléen exprimé sur les variables X de M. Une contrainte d'équité portant sur c exprime le fait que quelle que soit la séquence d'entrée σ_X de longueur infinie il n'existe aucun suffixe de longueur infinie de σ_X dont les entrées satisfont toutes $\bar{c}$. En d'autres termes, le prédicat c doit être vrai *infiniment souvent* le long de σ_X. En termes de la logique linéaire, cela s'écrit GFc.

Une spécification opérationnelle d'environnement est un modèle M_{env} employé en composition parallèle avec M, tel que $O_{env} \cap X \neq \emptyset$ et que $M||M_{env}$ ne contient pas de cycle combinatoire. Le comportement de M_{env} peut éventuellement être influencé par celui de M ; il est donc possible d'avoir : $X_{env} \cap O \neq \emptyset$.

On remarque que l'on dispose des mêmes moyens, que ce soit pour spécifier un système ou un environnement. Cependant, dans le cas d'un environnement, il existe des modalités d'écriture qui sont préférables. Comme il a été remarqué dans cette même section, un prédicat spécifiant un environnement peut référencer des variables de sortie du système. Il peut ainsi restreindre les états atteignables du système ce qui est souvent contraire à la réalité et dangereux. On peut classer les modalités de spécification par ordre de préférence vis-à-vis de ce risque :

1. écrire une spécification opérationnelle M_{env} de l'environnement et la composer avec M ;
2. écrire une spécification logique de l'environnement ayant la forme suivante : $G(x_i \Leftrightarrow f_{env}(\langle x_{m-1}\rangle, \langle s_n\rangle, \langle o_p\rangle, \langle ND_r\rangle))$. La i-ème variable d'entrée a une valeur qui peut dépendre des autres variables d'entrée $x_1 ... x_{i-1}, x_{i+1}, ... x_m$, des variables d'état $\langle s_n\rangle$ et de sortie $\langle o_p\rangle$ du système, et éventuellement d'autres variables caractérisant le non-déterminisme de l'environnement modélisé. Dans le cas de l'expression d'une contrainte d'équité, on remplace dans l'expression ci-dessus l'opérateur temporel G par l'opérateur GF ;
3. écrire une spécification logique d'environnement ayant une forme arbitraire.

2.4 Algorithmes de preuve

Le problème qui consiste à déterminer si un modèle symbolique M satisfait sa spécification P (opérationnelle ou logique) est connu sous le nom de *vérification symbolique de modèles*[5].

Plusieurs méthodes permettent d'évaluer l'affirmation $M \models P$. Selon la nature de la spécification, (opérationnelle, logique arborescente ou linéaire) cette évaluation peut se faire en parcourant l'espace d'états du modèle M "en avant" depuis son état initial, par

[5]de l'anglais : *symbolic model checking*. Ce terme dénote la technique qui emploie des algorithmes symboliques pour la vérification de modèles décrits de manière symbolique.

Procedure parcours en avant(T, χ_{s_0}, P)
  atteint $\leftarrow \chi_{s_0}$; ancien $\leftarrow \chi_{s_0}$; frontière $\leftarrow \chi_{s_0}$;
  while (frontière $\neq 0$) et (frontière $\rightarrow$ P) **do**
    nouveau $\leftarrow \chi_{\text{IMG}(T,ancien)}$;
    frontière $\leftarrow \overline{\text{atteint}}$. nouveau ;
    atteint $\leftarrow$ atteint + nouveau ;
    ancien $\leftarrow$ frontière ;
  end while

Algorithme 2.1: Algorithme générique de parcours "en avant"

Procedure parcours en arrière(T, χ_{s_0}, P, type parcours)
  if type parcours = plus grand point fixe **then**
    ancien $\leftarrow 1$; nouveau $\leftarrow 1$; $Op \leftarrow$ "." ;
  else
    ancien $\leftarrow 0$; nouveau $\leftarrow 0$; $Op \leftarrow$ "+" ;
  end if
  repeat
    ancien $\leftarrow$ nouveau ;
    nouveau $\leftarrow$ P Op $\chi_{\text{PREIMG}(T,ancien)}$;
  until (nouveau = ancien)

Algorithme 2.2: Algorithme générique de parcours "en arrière"

une suite de calculs d'image, ou "en arrière" depuis l'ensemble de tous ses états potentiels, grâce au calcul de pré-image. Lors de cette évaluation, le modèle symbolique M est identifié par sa relation de transition T et par la fonction caractéristique de son état initial χ_{s0}.

Le parcours "en avant" a pour but de vérifier si à chaque pas l'ensemble des états atteints satisfont la spécification P et de s'arrêter dès l'instant où au moins un état atteint ne satisfait pas P. Si l'ensemble des états atteignables de M a été parcouru et que P n'a jamais été fausse alors $M \models P$. Ce parcours est résumé par l'algorithme 2.1.

Quant au parcours "en arrière", il démarre à partir de la globalité des états de M et effectue une suite d'itérations dans le but de calculer un sous-ensemble d'états satisfaisant P. Si l'état initial s_0 de M se trouve parmi ces états, alors $M \models P$. Ce parcours est résumé par l'algorithme 2.2 ci-dessus. Grâce au parcours en arrière, il est possible de calculer aussi bien un *plus petit point fixe*, qu'un *plus grand point fixe*, selon la spécification à évaluer. Cette distinction sera faite au paragraphe §2.4.2.1.

2.4.1 Restriction du modèle symbolique pour la preuve

Une étape très importante, qui doit précéder tout parcours symbolique, consiste à restreindre le modèle symbolique aux seules parties qui peuvent avoir une influence sur le résultat (valeur logique) de la vérification. Ce sous-ensemble est connu sous le nom de *cône d'influence* d'une spécification [8]. Un cône d'influence est un sous-ensemble qui contient toutes les variables de M qui apparaissent dans la spécification à prouver, ainsi que toute variable dont la valeur peut avoir une influence (à travers les fonctions de transition ou de sortie) sur celles-ci. D'une manière plus intuitive, cette restriction peut être réalisée sur le réseau de portes qui représente le modèle à vérifier M : on démarre avec les variables figurant dans la spécification, et on procède à un coloriage des nœuds en remontant vers les entrées. Tout nœud du réseau déjà coloré est ignoré. Le sous-réseau coloré résultant représente le cône d'influence à utiliser pour la preuve. Nous identifions l'ensemble obtenu sous le nom de *cône d'influence structurel*.

Cependant, comme il sera illustré sur l'exemple de la ré-implémentation modulaire du modèle ARB' §2.4.3.4, le cône structurel possède quelques limites : la réduction conforme au graphe de dépendance n'est pas optimale. La présence d'une variable dans le cône d'influence n'implique pas son influence logique sur l'expression analysée.

Les BDDs constituent un outil efficace pour l'optimisation de la taille du cône d'influence. Soit $f(v_1, v_2, \ldots v_k)$ une expression logique. L'expression booléenne de f peut avoir une forme quelconque, dans laquelle toutes les variables v_i apparaissent au moins une fois sous forme directe ou complémentée. En revanche, le BDD qui représente f fera apparaître les seules variables dont la valeur influe sur la valeur de f. Donc, si une variable v_j n'apparaît pas dans la liste de variables du BDD, cela signifie que sa valeur peut être ignorée.

On peut ainsi calculer le cône d'influence par analyse successive de la représentation par BDD de la formule qui constitue une spécification logique. Cette procédure est décrite par l'algorithme 2.3. On démarre avec un modèle symbolique identifié par un ensemble FT de fonctions de transition et un ensemble FS de fonctions de sortie. A partir de la formule temporelle F à prouver, on isole l'ensemble des prédicats booléens qui y sont contenus. Pour chaque prédicat booléen P contenu dans la formule à prouver, cet algorithme produit deux ensembles : "RESULTAT" et "Cône". L'ensemble "RESULTAT" contient les BDDs à utiliser pendant le parcours symbolique ; l'ensemble "Cône" contient toutes les variables d'entrée, d'état et de sortie dont dépend le prédicat P. L'algorithme utilise une structure de type file "FILE-A-TRAITER" avec des primitives pour ajouter (ENFILER) et enlever (DEFILER) des éléments, ainsi que pour tester si une file est vide. La fonction VARS appliquée à un BDD renvoie l'ensemble des variables dont dépend ce BDD. On emploie également la fonction $BDD(FT, FS, v)$, qui pour une variable v, renvoie soit une fonction de transition si $v \in S$, soit une fonction de sortie si $v \in O$. Cette fonction est appellée uniquement avec une variable d'état ou de sortie.

Le cône optimisé est l'union de tous les ensembles "Cône" obtenus.

La fonction caractéristique de l'état initial dans le modèle restreint s'obtient à partir de la fonction χ_0 du modèle de base, en gardant seulement les variables d'état présentes dans l'ensemble "Cône".

```
Procedure cône d'influence (FT, FS, F)
  FILE-A-TRAITER ← ∅ ;
  Cone ← ∅ ;
  RESULTAT ← ∅ ;
  /* Pour chaque prédicat booléen P contenu dans la formule F : */
  for P ∈ F do
    bddP ← ConstruireBDD(P) ;
    /* Pour chaque variable qui apparaît dans la formule P : */
    for v ∈ VARS(bddP) do
      Cone ← Cone ∪ v ;
      if v ∈ O ou v ∈ S then
        ENFILER(FILE-A-TRAITER, BDD(FT, FS, v)) ;
      end if
    end for
    while FILE-A-TRAITER non vide do
      q ← DEFILER(FILE-A-TRAITER) ; /* q est un BDD */
      RESULTAT ← RESULTAT ∪ q ;
      for v ∈ VARS(q) do
        Cone ← Cone ∪ v ;
        if v ∈ O ou v ∈ S then
          ENFILER(FILE-A-TRAITER, BDD(FT, FS, v)) ;
        end if
      end for
    end while
  end for
```

Algorithme 2.3: Algorithme de calcul d'un cône d'influence optimisé

Théorème 2.1 *Le modèle M satisfait la formule temporelle F sous restriction du cône d'influence optimisé si et seulement s'il la satisfait également sous restriction du cône structurel.*

Lemme 2.1 *Le cône optimisé est inclus dans le cône structurel.*

En effet toute variable $v \in Cone$ qui est une variable d'entrée ou d'état intervient à un moment donné dans au moins un BDD contenu dans l'ensemble "RESULTAT". Par conséquent, v figure nécessairement dans une des expressions booléenes dont dépend la formule F. Ce qui implique le fait que v appartient au cône structurel de F.

Lemme 2.2 *Toute variable qui appartient au cône structurel sans appartenir au cône optimisé n'a aucune incidence sur les valeurs des fonctions de transition et de sortie retenues dans l'ensemble "RESULTAT".*

En effet, si $v \in$ cône structurel $\setminus$ Cône, cela signifie qu'il ne s'est retrouvé en support d'aucun BDD de l'ensemble "RESULTAT". Par construction d'un BDD, une variable v' figure parmi les variables d'un BDD si et seulement si sa valeur a une incidence sur l'expression représentée par ce BDD. Ce n'est pas le cas de la variable v.

Compte-tenu de ce fait, la preuve $M \models F$ donnera le même résultat quelle que soit la restriction, structurelle ou optimisée qui est appliquée.

2.4.2 Preuve de spécifications logiques

Comme de nombreux travaux l'ont montré[21, 63, 30], toutes les formules temporelles, linéaires ou arborescentes, peuvent être exprimées grâce aux opérateurs propositionnels "ou" et "négation", ainsi qu'aux quantificateurs d'état "X", "G", "U" et de chemin "E". Comme dans le cas de la logique propositionnelle, les quantificateurs universels de chemins peuvent être réécrits en termes de quantificateurs existentiels de chemins. Il en est de même pour les quantificateurs d'états. Soient f et g deux formules temporelles CTL. Les règles de réécriture suivantes s'appliquent :

$$\begin{aligned}
f \,.\, g &\Leftrightarrow \overline{\bar{f} + \bar{g}} \\
AX\ f &\Leftrightarrow \overline{EX\ \bar{f}} \\
EF\ f &\Leftrightarrow E\ 1\ U\ f \\
AF\ f &\Leftrightarrow \overline{EG\ \bar{f}} \\
AG\ f &\Leftrightarrow \overline{EF\ \bar{f}} \\
A\ f\ U\ g &\Leftrightarrow \overline{E(\bar{g}\ U\ \overline{f + g}) + EG\ g}
\end{aligned}$$

Dans le cas où f et g sont deux formules temporelles LTL, les règles de réécriture suivantes s'appliquent :

$$\begin{aligned}
f \,.\, g &\Leftrightarrow \overline{\bar{f} + \bar{g}} \\
F\ f &\Leftrightarrow 1\ U\ f \\
G\ f &\Leftrightarrow \overline{F\ \bar{f}}
\end{aligned}$$

Soit f et g deux formules temporelles arborescentes, telles que g est une sous-formule de f. La formule f affirme la satisfaisabilité de g dans M à l'instant présent et/ou dans le futur. C'est pourquoi, pour tout état e de M tel que $e \models f$ on a : $e \models f$ et/ou il existe au moins un chemin partant de e et contenant un état e' tel que $e' \models g$. Pour calculer l'ensemble des états de M qui satisfont f, on démarre donc avec les états qui satisfont g et on calcule leurs prédécesseurs dans M. Ainsi, l'évaluation de f consiste à effectuer une recherche en arrière dans le graphe d'états de M. Cette méthode convient pour l'évaluation des formules CTL et peut être adaptée à l'évaluation des formules temporelles linéaires, comme il sera montré dans la suite de ce chapitre.

2.4.2.1 Preuve des formules CTL

Soit E un ensemble d'états dans M et $E_f \subseteq E$ le sous-ensemble de E satisfaisant la formule f. On note χ_f la fonction caractéristique représentant ce sous-ensemble. L'expression de χ_f peut prendre une parmi les formes suivantes[22] :

- si f est exprimée uniquement sur les variables $X \cup S \cup O$ alors $\chi_f = \chi_E \,.\, f$. Dans le cas particulier où E contient tous les états potentiels de M, $\chi_E = 1$ et $\chi_f = f$;
- si $f = \bar{g}$ alors $\chi_f = \bar{\chi}_g$;
- si $f = g + h$ alors $\chi_f = \chi_g + \chi_h$;
- si $f = EXg$ alors χ_f exprime l'ensemble d'états ayant au moins un successeur satisfaisant g : $\chi_f(\langle s_n \rangle) = \exists \langle x_m \rangle \exists \langle s'_n \rangle : T(\langle x_m \rangle, \langle s_n \rangle, \langle s'_n \rangle) \,.\, \chi_g(\langle s'_n \rangle)$
  χ_f est en réalité la pré-image de l'ensemble χ_g à travers la relation de transition T, $\chi_{\textsc{PreImg}(\chi_g, T)}$;
- si $f = EGg$ alors χ_f est le plus grand point fixe de l'équation :

$$Y = \chi_g \,.\, \chi_{\textsc{PreImg}(Y,T)}; \tag{20}$$

- si $f = E\ gUh$ alors χ_f est le plus petit point fixe de l'équation :
  $Y = \chi_h + (\chi_g \,.\, \chi_{\textsc{PreImg}(Y,T)})$

Une fois χ_f obtenue il reste à déterminer si l'état initial s_0 satisfait f : cet état doit appartenir à l'ensemble χ_f. Pour cela on vérifie que $\chi_f(\langle s_0 \rangle) = 1$.

Prise en compte des contraintes d'équité Une contrainte d'équité est un prédicat booléen c exprimé sur les variables $X \cup S \cup O$ permettant de spécifier un ensemble de séquences d'états de longueur infinie, le long desquelles c est satisfait infiniment souvent. Cette propriété est naturellement exprimée par la formule linéaire GFc. Une telle séquence d'états est dite *équitable* par rapport à c. L'adaptation à CTL du raisonnement sur des séquences équitables nécessite un quantificateur existentiel de chemin permettant de désigner des chemins équitables uniquement. Soit C_{eq} un ensemble contenant un nombre arbitraire de prédicats c constituant chacun une contrainte d'équité. Le quantificateur $E_{C_{eq}}$ affirme l'existence d'un chemin d'états $\mathbf{e}$ de longueur infinie, tel que les éléments de C_{eq} sont satisfaits infiniment souvent le long de $\mathbf{e}$. En d'autres termes, ceci signifie qu'il n'existe aucun suffixe de $\mathbf{e}$ de longueur infinie le long duquel un ou plusieurs éléments de C_{eq} ne sont jamais satisfaits :

$$\exists \mathbf{e} = (e_1, e_2, \ldots,) : \\ \forall k > 0 : \mathbf{e}_k = (e_k, e_{k+1}, \ldots,), \forall c \in C_{eq} : \mathbf{e}_k \not\models G\ \bar{c}$$

Compte-tenu de cette définition, l'adaptation des opérateurs CTL aux chemins équitables se fait de la manière suivante :

- si $f = E_{C_{eq}} Gg$ alors on note Y la fonction caractéristique d'un ensemble d'états tels que :
  1. chaque élément de Y satisfait g ;
  2. tout élément de Y se trouve au début d'un chemin équitable par rapport à chacun des éléments de C_{eq} et dont les états satisfont chacun g :

$$\mathcal{EQ} = \bigwedge_{c \in C_{eq}} E(gU(Y \,.\, c))$$

Ainsi, l'ensemble χ_f est le plus grand point fixe de l'équation :
$Y = \chi_g \cdot \chi_{EX(\mathcal{EQ})}$
- si $f = E_{C_{eq}} X g$ alors f peut se réécrire sous la forme $f = EX(g \,.\, E_{C_{eq}} G\ 1)$;
- si $f = E_{C_{eq}}(gUh)$ alors f peut se réécrire sous la forme $f = E(gU(h \,.\, E_{C_{eq}} G\ 1))$.

2.4.2.2 Exemple : preuve de propriétés de l'arbitre à trois entrées

Cette section illustre l'application des algorithmes de preuve à la vérification du modèle ARB' de l'arbitre symbolique à trois entrées.

Une parmi les propriétés importantes de ARB' décrit l'état de marche de celui-ci : *quelle que soit la séquence d'états démarrant à l'état initial de ARB', quel que soit l'état de cette séquence, si une ou plusieurs entrées sont activées, alors, au moins une variable de sortie prend la valeur 1.*

Cette condition de fonctionnement à un instant donné s'exprime à l'aide de l'expression :

$$S = (x_1 + x_2 + x_3) \rightarrow (o_1 + o_2 + o_3)$$

La suite de ce chapitre est consacrée à la comparaison de l'efficacité des différents algorithmes existants. Chaque algorithme de preuve est combiné avec une méthode de parcours symbolique de l'espace d'états de ARB, et le choix de cette méthode sera argumenté.

L'analyse du modèle symbolique ARB' et de S permet de construire le cône d'influence de ce prédicat. Pour cet exemple très simple, la restriction de ARB' par rapport au cône d'influence associé à S est ARB' lui même.

Spécification logique arborescente. L'expression CTL de la propriété "état de marche" de ARB est :

$$P_S = AG\ S$$

La preuve $ARB' \models P_S$ peut être abordée de deux façons distinctes : par évaluation d'un invariant (parcours en avant), ou par calcul de l'ensemble des états qui la satisfont (parcours en arrière).

Comme illustré au paragraphe §1.4.4.1, le parcours symbolique de ARB' peut bénéficier d'un ordre de variables selon lequel la quantification "au plus tôt" est efficace.

Preuve par évaluation d'invariant. L'opérateur temporel AG affirme que le prédicat S exprimant le fonctionnement de ARB' à un instant donné doit être satisfait dans tout état atteignable de ARB'. C'est pourquoi, la preuve $ARB' \models P_S$ s'effectue en calculant l'ensemble des états atteignables χ_{ATT} de ARB', grâce à l'exécution de l'algorithme :

$$\text{parcours en avant}(T, \chi_0, S)$$

Voici le déroulement de l'exécution de cet algorithme :

1. l'état initial de ARB' est caractérisé par la fonction $\chi_0 = (s_1, s_2, s_3) = \bar{s}_1.\bar{s}_2.\bar{s}_3$;
   $\text{atteint}_1 = \chi_0$, $\text{ancien}_1 = \chi_0$, $\text{frontière}_1 = \chi_0$;
   La représentation de atteint_1 nécessite 3 nœuds BDD. L'évaluation de l'expression $\text{frontière}_1 \rightarrow S$ résulte en une tautologie ;
2. $\text{nouveau}_2 = \underline{\chi_{\text{IMG}(T,ancien_1)}} = \bar{s}_1.\bar{s}_2 + \bar{s}_1.s_2.\bar{s}_3 + s_1.\bar{s}_2.\bar{s}_3$;
   $\text{frontière}_2 = \overline{\text{atteint}_1}.\text{nouveau}_2 = \bar{s}_1.\bar{s}_2.s_3 + \bar{s}_1.s_2.\bar{s}_3 + s_1.\bar{s}_2.\bar{s}_3$;
   $\text{atteint}_2 = \text{atteint}_1 + \text{nouveau}_2 = \text{nouveau}_2$;

La représentation de atteint_2 nécessite 4 nœuds BDD. L'évaluation de l'expression $\text{frontière}_2 \rightarrow S$ résulte en une tautologie ;

3. $\text{nouveau}_3 = \chi_{\text{IMG}(T,ancien_2)} = \text{nouveau}_2$;
   $\text{frontière}_3 = 0$; La condition d'arrêt du parcours symbolique est remplie.

Le prédicat S est vrai dans chaque état atteignable de ARB'. La preuve s'est déroulée en deux étapes de calcul effectif (calcul d'image).
□

Preuve par calcul de l'ensemble d'états satisfaisant P_S. Alternativement, on peut considérer la réécriture de P_S en utilisant les opérateurs temporels E et U :

$$P_S = \overline{EF\ \bar{S}} = \overline{E\ (1\ U\ \bar{S})}$$

La preuve de P_S nécessite le calcul de l'ensemble d'états potentiels de ARB', qui satisfont cette propriété. Cet ensemble est obtenu grâce à l'algorithme de parcours en arrière, permettant de calculer le plus petit point fixe :

$$\text{parcours en arrière}(T, \chi_0, bar\,S, 0)$$

Voici le déroulement de cet algorithme :

1. calcul de plus petit point fixe : $\text{ancien}_1 = 0$, $\text{nouveau}_1 = 0$, $Op =$ "+" ;
2. $\text{ancien}_2 = \text{nouveau}_1$; $\text{nouveau}_2 = \bar{S} + \chi_{\text{IMG}(T,ancien_1)} = \bar{S}$;
   $\text{ancien}_2 \neq \text{nouveau}_2$, donc le parcours continue. La représentation de nouveau_2 nécessite 12 nœuds BDD ;
3. $\text{ancien}_3 = \text{nouveau}_2$; $\text{nouveau}_3 = \bar{S} + \chi_{\text{IMG}(T,ancien_2)} = \bar{S}$; $\text{ancien}_3 = \text{nouveau}_3$, donc le point fixe a été atteint. La représentation de nouveau_3 nécessite 12 nœuds BDD ;

Le terme ancien_3 représente l'ensemble des états qui satisfont la propriété $E\ (1\ U\ \bar{S})$. L'ensemble d'états recherché est le complément de ancien_3. Reste donc à vérifier que l'état initial de ARB' en fait partie : $\chi_0 \rightarrow \overline{\text{ancien}_3}$. Ce prédicat est vrai.

Une approche alternative de calcul de l'ensemble d'états satisfaisant P_S utilise également le parcours en arrière, mais en s'appuyant sur le fait que le résultat est nécessairement inclus dans l'ensemble représenté par atteint_2. La preuve démarre en calculant l'ensemble des états atteignables de ARB', et en obtenant le prédicat $\chi_{ATT} = \text{atteint}_2$, qui caractérise cet ensemble. Ensuite, on recherche le plus petit point fixe de l'équation :

$$Y = P\ .\ \chi_{ATT} + \chi_{\text{PREIMG}(Y,T)}$$

où $P = S$, la condition de fonctionnement de ARB'. Une fois χ_{ATT} calculé, le parcours symbolique en arrière est similaire à celui présenté ci-dessus. Le résultat est obtenu en un nombre d'itérations identique. On note toutefois pour cet exemple une nette amélioration de la taille des termes manipulés durant la preuve : les termes nouveau_2 et nouveau_3 valent tous les deux 0. Ces différences seront approfondies au paragraphe §2.5 - Discussion.
□

2.4.2.3 Preuve des formules LTL

Toute formule LTL se présente sous la forme $f = A\ g$ où g représente une formule de chemin. Les éventuelles sous-formules d'état qui composent g ne peuvent être que des

propositions atomiques. La formule f peut être réécrite grâce à l'opérateur de chemin "E" : $f = \overline{E\ \bar{g}}$. Ainsi, pour évaluer toute formule linéaire, il suffit de pouvoir évaluer des formules de la forme Eh, où $h = \bar{g}$ représente une formule de chemin obéissant aux mêmes restrictions que g. Un modèle M satisfait la formule f si et seulement si tous les chemins d'états (ou traces) possibles de M qui démarrent à l'état initial satisfont g. En d'autres termes, M ne doit inclure aucune trace satisfaisant h.

La méthode de preuve classique pour les formules linéaires s'appuie sur le concept d'*inclusion des traces*. On construit un modèle symbolique M_h[6] tel que toute séquence d'états satisfaisant h doit être contenue dans M_h. Le modèle M_h est ensuite composé avec M. Le modèle résultant représente l'*intersection* en termes de traces des modèles M_h et M. Si le résultat de cette intersection est vide, alors M et M_h n'ont aucune trace en commun et donc $M \models f$.

Une méthode possible de construction de M_h est présentée dans [55]. Elle est constituée de plusieurs étapes :

1. réécriture de h en termes des opérateurs "négation", "X" et "U" ;
2. décomposition de h en sous-formules élémentaires. Chaque sous-formule élémentaire est en fait une formule temporelle linéaire. Le but de cette décomposition est de *dérouler* h de manière à rendre explicites les relations état courant - prochain état. Chaque sous-formule de h est mise en correspondance avec un ou plusieurs états de M_h où elle est satisfaite.
   L'ensemble $fe(h)$ des formules élémentaires de h est constitué des éléments suivants :
   - $fe(1) = fe(0) = \emptyset$;
   - $fe(p) = \{p\}$ si $p \in X \cup S \cup O$;
   - $fe(\bar{p}) = fe(p)$;
   - $fe(p + q) = fe(p) \cup fe(q)$;
   - $fe(Xp) = \{X\ p\} \cup fe(p)$;
   - $fe(pUq) = \{X(p\ U\ q)\} \cup fe(p) \cup fe(q)$.

   On construit la machine M_h de façon à ce que chaque état associe une valeur logique aux éléments de $fe(h)$. Ainsi, M_h aura $nh = |fe(h)|$ variables d'état, permettant d'encoder toutes les valeurs logiques possibles pour les éléments de $fe(h)$;
3. construction de l'ensemble S_h des variables d'état de M_h :
   $S_h = \{p \mid p \in fe(h) \text{ et } p \in X \cup S \cup O\} \ \cup \ \{v_{Xp} \mid Xp \in fe(h)\}$
   Ainsi, l'ensemble S_h contient une partie des variables de M contenues dans la formule f et associe une variable booléenne à chaque élément de la forme Xp de $fe(h)$;
4. calcul de la fonction caractéristique de l'ensemble d'états de M_h qui satisfont la formule h. Les trois premières règles de calcul présentées au paragraphe §2.4.2.1 restent appliquables dans ce contexte. Les règles de calcul spécifiques concernent les états satisfaisant les sous-formules de la forme Xp et pUq :
   - $\chi_{Xp} = v_{Xp}$, où $v_{Xp} \in S_h$ est une variable d'état dont la valeur est 1 dans tous les états de M_h qui satisfont la sous-formule Xp ;
   - $\chi_{pUq} = \chi_q + (\chi_p \cdot \chi_{XpUq})$. La fonction caractéristique χ_{XpUq} est calculée conformément à la règle énoncée précédemment : $\chi_{XpUq} = v_{XpUq}$ où $v_{XpUq} \in S_h$.

[6] également connu sous le nom de *tableau*

5. calcul de la relation de transition T_h de M_h. Chaque sous-formule de la forme $Xp \in fe(h)$ est vraie dans un état donné si et seulement si p est vraie dans tous les successeurs de cet état :

$$T_h(\langle v_{nh} \rangle, \langle v'_{nh} \rangle) = \bigwedge_{v_{Xp} \in S_h \mid X\ p \in fe(h)} v_{Xp} \Leftrightarrow \chi_p(\langle v'_{nh} \rangle)$$

6. choix de l'état initial de M_h : $\chi_0 = \chi_h$.

Par construction de M_h, quelle que soit la sous-formule de la forme pUq apparaissant dans h, tout état e tel que $e \models pUq$ possède un successeur e' tel que $e' \models pUq$ ou $e' \models q$. Ainsi, des traces ayant un suffixe infini dont les états satisfont pUq sans jamais satisfaire q sont possibles, ce qui contredit la définition de l'opérateur linéaire U. Ces traces sont donc à exclure du fonctionnement de M_h : pour chaque sous-formule pUq de h, on retient les traces démarrant à l'état initial de M_h, telles que si pUq est vraie dans un état e de M_h alors il existe nécessairement un état e' successeur de e tel que $e' \models q$. Cette restriction s'exprime grâce à un ensemble de contraintes d'équité :

$$C_{eq} = \{\overline{\chi_{pUq}} + q \mid \text{ pour chaque sous-formule de la forme } pUq \text{ apparaissant dans } h\}$$

La preuve $M \models f$ consiste à chercher l'ensemble d'états de $M||M_h$ qui constituent le début d'une trace qui satisfait infiniment souvent chaque contrainte d'équité de l'ensemble C_{eq}. Cet ensemble est caractérisé par la formule CTL : $E_{C_{eq}} G\ 1$. Si l'état initial de $M||M_h$ appartient à cet ensemble, alors M et M_h possèdent une trace en commun et cette trace satisfait h, violant ainsi f. Dans le cas contraire, $M \models f$.

Techniques avancées de construction de tableaux. La technique de construction de tableaux présentée ci-dessus n'est pas adaptée lorsque la taille de la formule linéaire à traduire augmente. En effet, plus il y a d'éléments dans l'ensemble $fe(h)$, et plus la représentation symbolique de M_h sera coûteuse en nombre de variables d'état. D'autre part, l'emploi de l'algorithme de preuve qui calcule l'ensemble d'états satisfaisant la formule $EG\ 1$ sous hypothèse de contraintes d'équité est une opération très coûteuse, car elle nécessite un calcul de point fixe imbriqué : chaque itération de point fixe est elle-même un calcul de point fixe.

Cette technique a été raffinée dans [41] par Gerth, Peled et al. Les auteurs proposent une méthode de construction de tableaux dont la taille (nombre d'états) est plus compacte. Cependant, le tableau résultant est représenté de manière explicite, et la vérification se fait également selon un algorithme explicite. L'impact du gain apporté par cette méthode sur l'efficacité de la preuve *symbolique* de formules LTL est mineur. Le fait de supprimer un certain nombre d'états d'une machine n'entraine pas d'amélioration notable sur la taille de sa représentation symbolique. Cette même technique a été reprise par la suite [28], dans le but d'optimiser la taille du tableau à travers des méthodes syntaxiques de manipulation des formules.

Une contribution très importante a été apportée par [68, 58]. Les auteurs proposent un classement des formules LTL selon un critère de *force*, directement lié à la complexité

de preuve de la formule sous-jacente. La notion de force fait référence au caractère de l'opérateur temporel pUq. Par définition, cet opérateur est *fort*[7], car le prédicat q doit nécessairement se produire. Une version faible du même opérateur n'inclut pas cette nécessité.

Le critère de force est repris et raffiné par Somenzi et al. dans [12]. Selon leur classification, une formule LTL peut engendrer un tableau *fort*, *faible* ou *terminal*. Ces classes sont caractérisées de manière structurelle. On divise l'ensemble des états de M_h en k partitions, $E_1, \ldots E_k$. Un ordre partiel ($\leq$) est construit sur ces partitions : si $E_i \leq E_j$ alors il existe deux états $e \in E_i$ et $e' \in E_j$ tels que la transition $e \rightarrow e'$ soit possible au sein du tableau. On note $\mathcal{EQ}$ l'ensemble des états caractérisé par $\chi_{C_{eq}}$. Dans ce cadre, un tableau est "faible" si et seulement si :

$$\forall i = 1..k : E_i \subseteq \mathcal{EQ} \text{ ou } E_i \cap \mathcal{EQ} = \emptyset$$

Les tableaux "terminaux" constituent un cas particulier des tableaux faibles. Si toutes les partitions E_i telles que $E_i \subseteq \mathcal{EQ}$ sont des éléments maximaux de l'ordre partiel (il n'existe pas de partition E_j telle que $E_i \leq E_j$), alors le tableau est terminal.

La différence essentielle entre ces classes réside dans l'algorithme symbolique nécessaire pour vérifier si un ensemble de traces est vide ou non [12] :

- dans le cas d'un tableau M_h fort, la preuve nécessite l'évaluation de la formule $E_{C_{eq}} G\ 1$ sur le modèle $M||M_h$;
- si le tableau est faible, alors les seules séquences d'états acceptables de M_h doivent être équitables par rapport à l'ensemble d'états $\mathcal{EQ}$. Les états de M_h sont regroupés en $E_1, \ldots E_k$, partitions partiellement ordonnées. Ainsi, lorsqu'une transition de M_h quitte la partition E_i, il n'est plus possible d'y retourner. Chaque séquence d'états doit donc atteindre une partition $E_j \subseteq \mathcal{EQ}$ et ne la quitte que pour se diriger vers une autre partition $E_j \subseteq \mathcal{EQ}$. Comme le nombre des partitions est fini et qu'un retour vers l'arrière n'est pas possible, la seule façon de satisfaire la condition d'équité est d'atteindre inévitablement une telle partition et d'y rester indéfiniment. Une telle séquence d'états est caractérisée par la formule CTL $EF\ EG\chi_{C_{eq}}$.
  Pour la preuve, il suffit donc d'évaluer la formule CTL $\overline{EF\ EG\chi_{C_{eq}}}$ sur le modèle $M||M_h$;
- enfin, dans le cas d'un tableau terminal, toute séquence d'états équitable doit atteindre une partition $E_i \subseteq \mathcal{EQ}$. Etant un élément maximal de l'ordre partiel, cette partition ne pourra plus être quittée. Ces séquences sont caractérisées par la formule CTL $EF\ \chi_{C_{eq}}$.
  Pour la preuve, il suffit d'évaluer la formule CTL $\overline{EF\ \chi_{C_{eq}}}$ sur le modèle $M||M_h$.

Le tableau 2.3 montre des exemples de formules temporelles linéaires appartenant à chacune des classes de tableaux. Chaque fois qu'il est possible, un tableau "faible" est généré, ce qui permet d'éviter l'utilisation systématique de l'algorithme de preuve $E_{C_{eq}} G\ 1$ au profit d'algorithmes plus performants. Cette différence est très importante et sera illustrée dans l'exemple présenté au paragraphe §2.4.2.4.

Dans [80], Somenzi et Bloem présentent une technique de génération plus pointue. La maîtrise de la taille du tableau résultant se fait à la fois en appliquant des règles de

[7] par contraste par rapport à la version *faible* : $pWq = pUq + G\ p$

Force	Exemple LTL
fort	$G\ F\ p$
	$G\ F\ p \rightarrow G\ F\ q$
faible	$F\ G\ p$
	$G\ F\ p \rightarrow F\ G\ q$
terminal	$F\ f$

TAB. 2.3 – Exemples de formules LTL appartenant aux classes de tableaux forts, faibles et terminaux

réécriture à la formule LTL à traduire, et en appliquant des techniques de minimisation logique sur le résultat.

2.4.2.4 Exemple : Preuve d'une spécification logique linéaire

L'expression LTL de la propriété "état de marche" de ARB' est :

$$P_S = G\ S$$

La preuve $ARB' \models P_S$ peut elle aussi être abordée de deux façons distinctes : par construction d'un tableau selon la méthode classique, et emploi systématique de l'algorithme $E_C G\,1$, ou par construction sélective d'un tableau de *force* minimale, accompagnée du choix d'un algorithme de preuve adapté. En tout état de cause, on construit un tableau associé à la négation de P_S. Ce tableau est ensuite composé avec ARB' afin de déterminer, par application de l'algorithme symbolique adéquat si l'intersection en termes de traces de ces deux modèles est vide.

L'ensemble des formules élémentaires pour cette spécification contient un nombre d'éléments assez élevé, ce qui entraîne un tableau très complexe. Dans un souci de clarté, nous allons recourrir à l'artifice suivant. Soit ARB'' un modéle symbolique identique à ARB' sauf en ce qui concerne les variables de sortie : $O'' = \{o_1, o_2, o_3, S\}$. On a $ARB' \models P_S$ si et seulement si $ARB'' \models P_S$.

Méthode classique. La construction de M_h (où $h = \bar{P}_S$) suit les étapes suivantes :

- réécriture de h en termes des opérateurs "X" et "U". On a $h = \overline{G\ S} = F\ \bar{S} = (1\ U\ \bar{S})$;
- calcul de $fe(h)$:
  $fe(h) = fe(1\ U\ \bar{S}) = \{X(1\ U\ \bar{S})\} \cup fe(1) \cup fe(\bar{S})$
  $= \{X(1\ U\ \bar{S})\} \cup fe(S) = \{X(1\ U\ \bar{S}), S\}$
- construction de l'ensemble de variables d'état de M_h. Cet ensemble associe une variable d'état à chaque élément de $fe(h)$: $S_h = \{v_{X(1\ U\ \bar{S})}, S\}$. Pour des raisons de lisibilité, on abrège $v_{X(1\ U\ \bar{S})}$ à v ;
- calcul de la fonction caractéristique $\chi_{1\ U\ \bar{S}}$:
  $\chi_{1\ U\ \bar{S}} = \chi_{\bar{S}} + \chi_{X(1\ U\ \bar{S})}$;
  où $\chi_{\bar{S}} = \bar{S}$ et $\chi_{X(1\ U\ \bar{S})} = v$; donc, $\chi_{1\ U\ \bar{S}} = \bar{S} + v$
- calcul de T_h la relation de transition de M_h :
  $T_h = (v \Leftrightarrow \chi_{1\ U\ \bar{S}}) = (v \Leftrightarrow (\bar{S}' + v'))$, où S' et v' sont les variables prochain état de S et v.

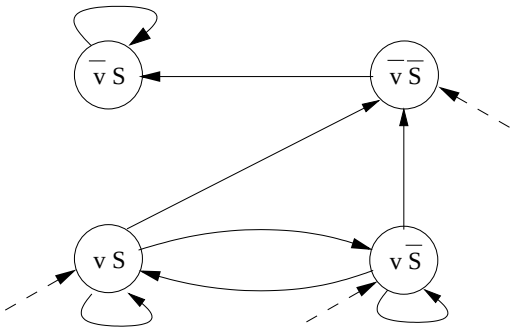

FIG. 2.3 – Graphe d'états de M_h, méthode de construction classique

– choix de l'état initial de M_h : $\chi_{h0} = \chi_{1\ U\ \bar{S}} = \bar{S} + v$.

Le graphe d'état de M_h est donné dans la figure 2.3.

A présent, construisons le modèle $M = M_h||ARB''$, tel que :

– $\chi_{M0} = \chi_{h0} \cdot \chi_0$;
– $T_M = T_h \ . \ T$;
– toutes les traces de M doivent être équitables par rapport à l'ensemble d'états caractérisé par $\chi_{eq} = \bar{\chi}_h + \chi_{\bar{S}} = \overline{(\bar{S} + v)} + \bar{S}$;

Voici les pas de calcul de l'ensemble des états χ_h de M satisfaisant la formule $E_{C_{eq}}G$ 1. Cet ensemble constitue le plus grand point fixe de l'équation 20.

1. calcul de l'ensemble d'états satisfaisant $EX\ E(1\ U\chi_{eq})$:
   Calcul des états satisfaisant $E(1\ U\chi_{eq})$ (plus petit point fixe) :
   – $\mathrm{n}_1 = \chi_{eq} + \chi_{\mathrm{PREIMG}(0,T_M)} = \chi_{eq}$;
   – $\mathrm{n}_2 = \chi_{eq} + \chi_{\mathrm{PREIMG}(\mathrm{n}_1,T_M)} = \mathrm{n}_1$. Le point fixe est atteint. La représentation de n_1 et n_2 nécessite 13 nœuds de BDD.

   $\mathrm{nouveau}_1 = \chi_{\mathrm{PREIMG}(\mathrm{n}_2),T_M} = v$;
2. calcul de l'ensemble d'états satisfaisant $EX\ E(1\ U\mathrm{nouveau}_2 \ . \ \chi_{eq})$:
   Calcul des états satisfaisant $E(1\ U\mathrm{nouveau}_1 \ . \ \chi_{eq})$ (plus petit point fixe) :
   – $\mathrm{n}_1 = \mathrm{nouveau}_1 \ . \ \chi_{eq} + \chi_{\mathrm{PREIMG}(0,T_M)} = v.\chi_{eq}$;
   – $\mathrm{n}_2 = \mathrm{nouveau}_1 \ . \ \chi_{eq} + \chi_{\mathrm{PREIMG}(\mathrm{n}_1,T_M)} = \mathrm{n}_1$. Le point fixe est atteint. La représentation de n_1 et n_2 nécessite également 13 nœuds de BDD.

   $\mathrm{nouveau}_2 = \chi_{\mathrm{PREIMG}(\mathrm{n}_2,T_M)} = 0$. Le plus grand point fixe est donc 0.

Ainsi, aucun état de M ne constitue le début d'une séquence équitable par rapport à χ_{eq}. Ceci signifie qu'il n'est pas possible d'amener M dans un état où le prédicat S est faux. Cet algorithme a nécessité un total de 6 pas de calcul de pré-image. Chaque itération de calcul du résultat final a nécessité le calcul local d'un plus petit point fixe.
□

Calcul d'un tableau réduit. La même formule linéaire engendre le tableau réduit M'_h de la figure 2.4. Ce tableau possède un état en moins et a été obtenu par un calcul de la plus grande bi-simulation sur M_h. En effet, on observe que les états étiquetés vS et $v\bar{S}$ sont équivalents, car ils appartiennent à l'ensemble initial de M_h et leurs ensembles de

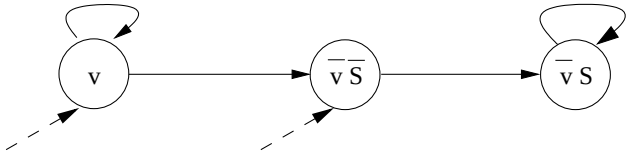

FIG. 2.4 – Graphe d'états du tableau simplifié M'_h

successeurs sont identiques. Ils peuvent de ce fait être fusionnés. En examinant le graphe d'état de M'_h on observe qu'il est possible de diviser l'ensemble de ses états en trois composantes fortement connexes totalement ordonnées. L'élément maximal de cet ordre est un état "puits" qui, une fois atteint, satisfait la condition d'équité C_{eq} infiniment souvent. Le modèle M'_h appartient à la classe des tableaux terminaux. Donc, prouver que l'ensemble des traces possibles de $M = M'_h||ARB''$ est vide revient à évaluer la formule CTL $M \models \overline{EF\chi_{eq}}$ grâce à un algorithme de plus petit point fixe.

Cet algorithme montre une utilisation de nœuds BDD similaire aux algorithmes précédents, mais il atteint le point fixe en seulement deux itérations.
□

2.4.3 Preuve de spécifications opérationnelles

2.4.3.1 Modèles abstraits : preuve de raffinement

Sous contrainte de correspondance totale entre les variables d'état de l'implémentation et du modèle abstrait, la preuve de raffinement procède en deux étapes :

1. calcul de l'ensemble d'états atteignables $\chi_{ATT}(\langle s_n \rangle)$ de *l'implémentation* M_{impl}, par calcul successif des termes χ_{S_i}, conformément à l'équation 8 ;
2. évaluation de l'expression :

$$\chi_{ATT}(\langle s_n \rangle) \rightarrow \exists \langle ND_r \rangle : \forall i \in \{1..n\} : F^i_{impl}(\langle x_m \rangle, \langle s_n \rangle) \Leftrightarrow F^i_{abs}(\langle x_m \rangle, \langle ND_r \rangle, \langle s_n \rangle) \tag{21}$$

On obtient que $M_{impl} \preceq_r M_{abs}$ si et seulement si cette expression est une tautologie.

Une optimisation immédiate de cet algorithme consiste à évaluer les expressions (21) pour chaque ensemble χ_{S_i} atteint lors d'une nouvelle itération de calcul de χ_{ATT}. Le parcours de l'espace d'états de M_{impl} s'arrête dès que l'évaluation ne résulte pas en une tautologie.

La preuve symbolique de raffinement constitue une technique relativement puissante pour évaluer une implémentation par rapport à sa spécification. Son efficacité réside dans la démarche compositionnelle sous-jacente qui est appliquable naturellement. Cet aspect sera illustré dans le chapitre 3 - Stratégies de preuve.

2.4.3.2 Exemple : preuve de raffinement par rapport à un modèle abstrait

Ecrivons une spécification opérationnelle ARB_{abs} de ARB', dont le but est de mettre en évidence l'aspect *exclusion mutuelle*. Il s'agit en fait de modéliser ARB' de manière

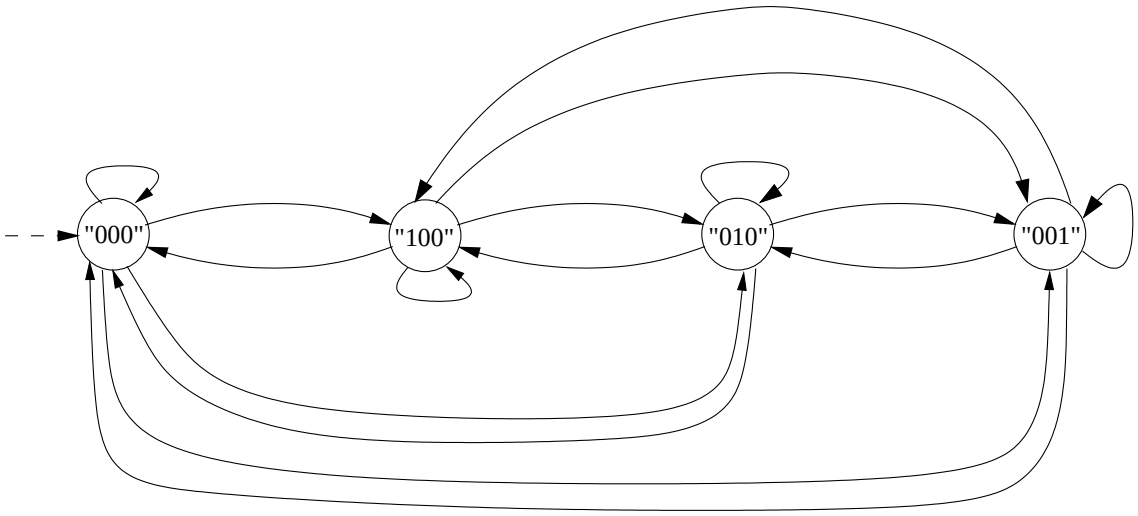

FIG. 2.5 – Graphe d'états du modèle abstrait de l'arbitre à trois entrées

abstraite, en se focalisant sur la fonctionnalité, dans notre cas l'exclusion mutuelle, sans se préoccuper des "détails" de l'implémentation. Un graphe d'états possible de ce modèle abstrait serait celui présenté dans la figure 2.5. L'unique souhait du concepteur a été de s'assurer qu'à tout moment, au plus une variable d'état du modèle abstrait prenait la valeur 1.

Les étiquettes associées aux états représentent les valeurs acceptables des variables d'état s_1, s_2 et s_3. Les transitions se font de manière non-déterministe. Pour décrire ce non-déterminisme, deux variables d'entrée auxiliaires (pseudo-entrées) ND_1 et ND_2 sont nécessaires. Ces variables permettent de choisir à un moment donné la transition menant vers un des quatre états de l'arbitre abstrait. Voici le modèle symbolique de ARB_{abs} :

- ensemble des variables d'entrée : $X_a = \{ND_1, ND_2\}$;
- ensemble des variables d'état : $S_a = \{s_1, s_2, s_3\}$;
- ensemble des variables de sortie : $O_a = \emptyset$;
- fonctions de transition :
  $Fa^1(s_1, s_2, s_3, ND_1, ND_2) = \overline{ND_1}.ND_2$;
  $Fa^2(s_1, s_2, s_3, ND_1, ND_2) = ND_1.\overline{ND_2}$;
  $Fa^3(s_1, s_2, s_3, ND_1, ND_2) = ND_1.ND_2$;
- L'état initial de l'arbitre abstrait est caractérisé par : $\chi_0(s_1, s_2, s_3) = \bar{s}_1.\bar{s}_2.\bar{s}_3$

Preuve de raffinement. La preuve $ARB' \preceq_r ARB_{abs}$ utilise l'ensemble d'états atteignables de ARB' calculé précédemment. L'algorithme de preuve est très semblable à celui de la preuve d'invariant. En reprenant et adaptant les mêmes étapes de calcul, on obtient :

1. l'état initial de ARB' est caractérisé par la fonction $\chi_0 = (s_1, s_2, s_3) = \bar{s}_1.\bar{s}_2.\bar{s}_3$; frontière$_1 = \chi_0$
   On a :
   $\chi_0 \rightarrow \exists ND_1, \exists ND_2 : ((Fa^1 \Leftrightarrow F_t^1).(Fa^2 \Leftrightarrow F_t^2).(Fa^3 \Leftrightarrow F_t^3))$, ainsi que

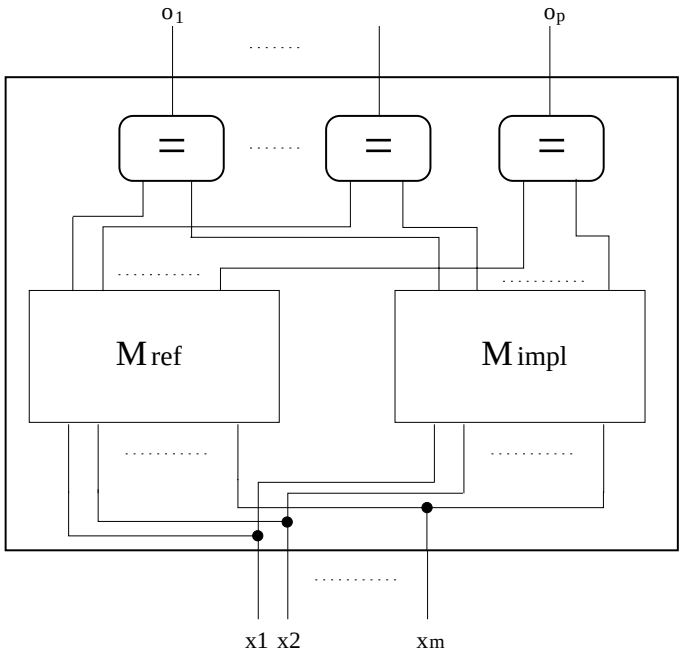

FIG. 2.6 – Produit des machines M_{ref} et M_{impl} pour la preuve d'équivalence

frontière$_1 \rightarrow \exists ND_1, \exists ND_2 : ((Fa^1 \Leftrightarrow F_t^1).(Fa^2 \Leftrightarrow F_t^2).(Fa^3 \Leftrightarrow F_t^3))$;

2. frontière$_2 = \bar{s}_1.\bar{s}_2.s_3 + \bar{s}_1.s_2.\bar{s}_3 + s_1.\bar{s}_2.\bar{s}_3$;
   On a :
   frontière$_2 \rightarrow \exists ND_1, \exists ND_2 : ((Fa^1 \Leftrightarrow F_t^1).(Fa^2 \Leftrightarrow F_t^2).(Fa^3 \Leftrightarrow F_t^3))$;
3. frontière$_3 = 0$; la condition d'arrêt du parcours symbolique est remplie.

2.4.3.3 Modèles de référence : preuve d'équivalence

Sous contrainte de correspondance totale entre les variables d'entrée et de sortie des modèles de référence et d'implémentation, la preuve d'équivalence séquentielle procède en trois étapes :

1. construction de la composition parallèle $M = M_{ref}||M_{impl}$, tout en respectant la correspondance des entrées et des sorties des deux modèles comparés (Figure 2.6) ;
2. calcul de l'ensemble d'états atteignables $\chi_{ATT}(\langle s_{n_1+n_2} \rangle)$ de M ;
3. pour chaque $i \in \{1..p\}$ vérifier que $\chi_{ATT} \rightarrow (o_i = 1)$;

Comme pour le cas du raffinement, une démarche similaire d'optimisation du parcours symbolique consiste à évaluer l'expression ci-dessus au fur et à mesure, pour chaque ensemble χ_{S_i} atteint lors d'une nouvelle itération de calcul de χ_{ATT}. Le parcours de l'espace d'états de M s'arrête dès que l'évaluation ne résulte pas en une tautologie.

On remarque que la preuve d'équivalence séquentielle peut se traduire en termes logiques temporels : $M_{ref} \equiv M_{impl}$ si et seulement si $M \models P_{eq}$, où $P_{eq} = AG((o_1 = 1).(o_2 =$

$1)\ldots.(o_p = 1))$. En règle générale, la preuve de cet invariant est coûteuse, car par construction, le cône d'influence associé à cette formule contient la totalité des variables de M. Ce problème se traite à plusieurs niveaux.

Tout d'abord, ce but de preuve peut être décomposé en sous-buts distincts, qui peuvent être prouvés séparément. En effet, conformément au résultat établi par Graf[44], le raisonnement $M \models P_{eq}$ peut être décomposé de la manière suivante :

$P_{eq} = P^1_{eq} \,.\, P^2_{eq} \,.\, \ldots P^p_{eq}$ où :
$P^1_{eq} = AG(o_1 = 1),\ P^2_{eq} = AG(o_2 = 1)$ et $P^p_{eq} = AG(o_p = 1)$

$M \models P_{eq}$ si et seulement si :
$M \models P^1_{eq}$ et $M \models P^2_{eq}$ et $\ldots M \models P^p_{eq}$

Par application de cette règle de décomposition, la preuve d'équivalence séquentielle se traduit par p preuves indépendantes. Grâce à la décomposition, il est possible de bénéficier d'une éventuelle réduction du cône d'influence relative aux formules $P^1_{eq}, P^2_{eq} \ldots P^p_{eq}$.

D'autre part, la composition de deux modèles distincts, partageant uniquement les entrées possède des propriétés structurelles intéressantes. Examinons la matrice de dépendance de M, lors du pas i du parcours symbolique. Plus précisément, on se positionne dans le "pire" des cas, selon lequel chaque fonction de transition dépend de toutes les variables d'entrée de M et de toutes les variables d'état présentes dans le même modèle. Soit $SR = \{sr_1, \ldots, sr_{n_1}\}$ l'ensemble des variables d'état de M_{ref} et $SI = \{si_1, \ldots, si_{n_2}\}$ l'ensemble des variables d'état de M_{impl}. On note $Fref^i, i = 1..n_1$ les fonctions de transition de M_{ref} et $Fimp^i, i = 1..n_2$ les fonctions de transition de M_{impl}.

$$
D_i = \begin{array}{c}
\\
Fref^1 \\
Fref^2 \\
\vdots \\
Fref^{n_1} \\
Fimp^1 \\
Fimp^2 \\
\vdots \\
Fimp^{n_2} \\
\chi_{S_i}
\end{array}
\begin{array}{c}
\begin{array}{cccccccc} x_1 \ldots & & x_m & sr_1 \ldots & & sr_{n_1} & si_1 \ldots & si_{n_2} \end{array} \\
\left(\begin{array}{cccccccc}
1 & \ldots & 1 & 1 & \ldots & 1 & 0 & \ldots 0 \\
1 & \ldots & 1 & 1 & \ldots & 1 & 0 & \ldots 0 \\
\ldots & \ldots & & & & & & \\
1 & \ldots & 1 & 1 & \ldots & 1 & 0 & \ldots 0 \\
1 & \ldots & 1 & 0 & \ldots & 0 & 1 & \ldots 1 \\
1 & \ldots & 1 & 0 & \ldots & 0 & 1 & \ldots 1 \\
\ldots & \ldots & & & & & & \\
1 & \ldots & 1 & 0 & \ldots & 0 & 1 & \ldots 1 \\
0 & \ldots & 0 & 1 & \ldots & 1 & 1 & \ldots 1
\end{array}\right)
\end{array}
$$

La matrice de dépendance est relativement creuse. En plus, à l'exception des variables d'entrée, sa distribution est bloc-diagonale, ce qui évoque la possibilité de recourir à une partition de la relation de transition en vue du parcours symbolique. Cette technique devrait permettre de quantifier séparément et indépendamment les variables $\langle sr_{n_1} \rangle$ et $\langle si_{n_2} \rangle$.

Cependant, la présence de la fonction "état courant", χ_{S_i}, a tendance à "casser" cette propriété structurelle. La partition de la relation de transition n'est donc pas possible dans le cas général.

En revanche, il est souvent vrai que l'état initial de M se caractérise par un unique produit de variables d'état, complémentées ou non. Ainsi, l'état initial χ_0 de M peut s'écrire sous la forme :

$$\chi_0(\langle sr_{n_1}\rangle, \langle si_{n_2}\rangle) = \chi_r(\langle sr_{n_1}\rangle) \,.\, \chi_i(\langle si_{n_2}\rangle)$$

Dans cette situation, la matrice de dépendance peut être mise sous une forme bloc-diagonale. Donc, tenant compte de la formule de calcul de l'image (6), le premier pas de parcours symbolique peut bénéficier de la partition de la relation de transition, en s'écrivant sous la forme :

$$\chi_{S_1} = \exists\langle x_m\rangle : \big((\exists\langle sr_{n_1}\rangle : \prod_{j=1}^{n_1}(sr'_j \Leftrightarrow Fref^j) \,.\, \chi_r) \,.\, (\exists\langle si_{n_1}\rangle : \prod_{j=1}^{n_2}(si'_j \Leftrightarrow Fimp^j) \,.\, \chi_i)\big) \tag{22}$$

C'est pourquoi, au moins pour ce premier pas de calcul, il est tout à fait possible d'éviter la construction d'une relation de transition monolithique.

Dans le cas général, l'expérience montre qu'il est rarement possible de manipuler une fonction "état courant" afin de pouvoir se ramener à une matrice creuse à distribution bloc-diagonale. C'est dans cette situation que la décomposition disjonctive peut se montrer efficace. Ainsi, en appliquant la décomposition disjonctive (formule (1.4.4.2)) selon la variable d'entrée $x_k, k = 1..m$ de M, le calcul d'image lors de l'étape i est décomposé en deux sous-buts distincts. Les matrices de dépendance $D_i^{x_k+}$ et $D_i^{x_k-}$ associées à chacun de ces sous-buts ont un poids (nombre d'éléments à 1) strictement inférieur au poids de D_i. Ceci est dû d'une part au fait que la colonne relative à x_1 affiche la valeur 0 en chaque position, et d'autre part tous les termes de la forme $s_j.x_k$ où $s_j.\bar{x}_k$ se trouvent éliminés, et peuvent entraîner de ce fait la disparition de s_j dans au moins une parmi $D_i^{x_k+}$ et $D_i^{x_k-}$. Une telle manipulation peut engendrer des matrices de dépendance à distribution bloc-diagonale ou triangulaire, adaptées pour la partition de la relation de transition ou pour la quantification "au plus tôt".

2.4.3.4 Exemple : preuve d'équivalence séquentielle

A partir de la description de ARB' le concepteur souhaite écrire une nouvelle description ARB_{mod} qui, contrairement à ARB', soit modulaire : il souhaite pouvoir rajouter de nouveaux clients, tout en effectuant un minimum de modifications à sa description. Il a opté pour l'écriture d'un modèle, $CANAL$, effectuant l'arbitrage pour un canal d'entrée. Ce modèle serait répliqué autant de fois qu'il y aurait de clients à servir. La description résultante est un modèle hiérarchique contenant autant d'instances $CANAL$ que de clients à servir. Considérons donc un arbitre modulaire, composé de N canaux, numérotés de 1 à N.

L'architecture d'un $CANAL$ doit reproduire le schéma de priorité de ARB'. Ainsi, les règles suivantes s'appliquent :

- le canal i est *prioritaire* sur le canal $i+1$ s'il reçoit une requête ($x_i = 1$) et si au cours du cycle précédent il n'a pas reçu une autorisation d'accès ;
- le canal i est *admissible* par le canal $i+1$ si ce dernier n'a pas de requête en cours : $x_{i+1} = 0$;
- si le canal i devient prioritaire, il l'est par rapport à tous les canaux $j > i$;
- si le canal i n'est pas admis, aucun des canaux $j > i$ ne l'est.

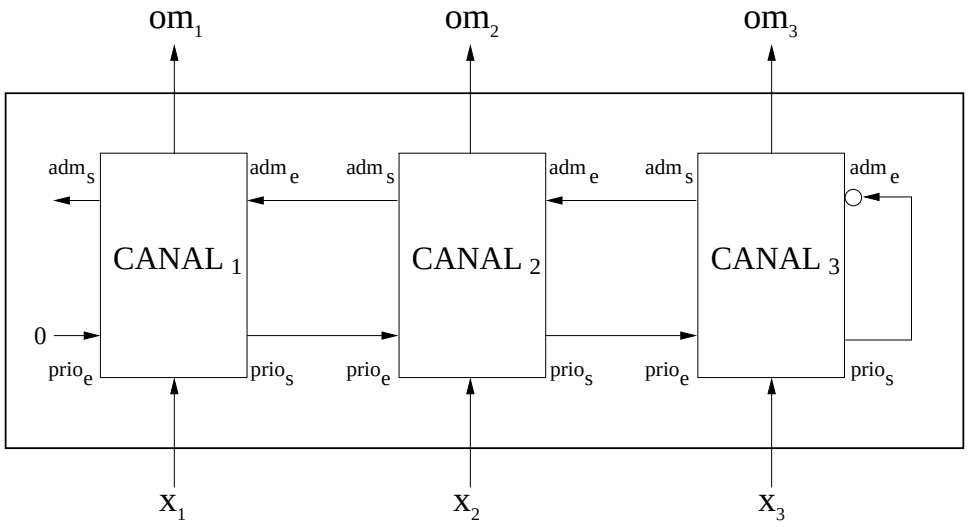

FIG. 2.7 – Arbitre à trois entrées obtenu par composition et interconnexion de trois instances $CANAL$

Le canal N n'a pas de successeur. Il est admissible si et seulement s'il n'est pas prioritaire par rapport à un éventuel successeur. Quant au canal 1, il n'a aucun prédécesseur prioritaire. Le canal i, ayant présenté une requête x_i, reçoit une permission d'accès si le canal $i-1$ n'est pas prioritaire et si le canal i est admissible ; de manière alternative, si le canal $i-1$ n'est pas prioritaire et si le canal i n'a pas précédemment bénéficié d'une permission d'accès, alors il reçoit une réponse positive.

La composition des modules $CANAL_i$ pour $N = 3$ est décrite par le schéma présenté figure 2.7.

Le premier canal est le plus prioritaire, sauf s'il vient de bénéficier d'une permission d'accès. Dans ce cas, ses requêtes sont acceptées seulement si aucun autre canal n'a de requête.

Le modèle symbolique d'un $CANAL_i$ est le suivant :

- ensemble des variables d'entrée : $Xm_i = \{x_i, prio_{ei}, adm_{ei}\}$;
- ensemble des variables d'état : $\{sm_i\}$;
- ensemble des variables de sortie : $Om_i = \{om_i, prio_{si}, adm_{si}\}$;
- fonction de transition : $sm'_i = Fm^i_t = x_i.\overline{prio_{ei}}.(adm_{ei} + x_i.\overline{sm_i})$;
- fonctions de sortie :
  $om_i = Fm^i_t$;
  $prio_{si} = prio_{ei} + x_i.\overline{sm_i}$;
  $adm_{si} = \bar{x}_i.prio_{ei}$;
- état initial : $\chi_{0i} = \overline{sm_i}$.

On obtient ainsi le modèle symbolique $ARB_{mod} = CANAL_1||CANAL_2||CANAL_3$.

Preuve d'équivalence. La preuve d'équivalence séquentielle entre ARB' et ARB_{mod} s'apparente à l'évaluation sur le modèle $ARB_{eq} = ARB'||ARB_{mod}$ de la formule temporelle CTL :

$$P_{eq} = AG((o_1 = om_1).(o_2 = om_2).(o_3 = om_3))$$

Ce but de preuve se décompose en sous-buts distincts :

$P_{eq} = P^1_{eq} \,.\, P^2_{eq} \,.\, P^3_{eq}$ où :

$P^1_{eq} = AG(o_1 = om_1)$, $P^2_{eq} = AG(o_2 = om_2)$ et $P^3_{eq} = AG(o_3 = om_3)$

Nous allons illustrer en détail la preuve formelle du premier but $P^1_{eq} = AG(o_1 = om_1)$. Les deux autres sont prouvés de manière similaire. La preuve combine une étape de restriction du modèle et une étape de parcours symbolique.

Restriction du modèle ARB_{mod}

Examinons tout d'abord le cône d'influence relatif à P^1_{eq}. En se basant sur les dépendances structurelles au sein de ARB_{eq}, le cône inclut les variables d'entrée x_1, x_2 et x_3, la variable d'état s_1 d'une part, et l'intégralité de ARB_{mod} d'autre part. En effet, la variable de sortie om_1 possède des dépendances structurelles envers chaque variable d'entrée et d'état de ARB_{mod}. Cependant, la représentation BDD de l'expression booléenne de om_1 dépend uniquement des variables d'entrée x_1, x_2, x_3 et d'état sm_1. Par application de l'algorithme d'optimisation du cône d'influence (2.3), il faut examiner la fonction de transition Fm^1_t de sm_1. Bien que cette fonction soit structurellement plus complexe, son BDD ne dépend que des variables x_1, x_2, x_3 et sm_1. Le cône d'influence optimisé contient donc les variables suivantes : x_1, x_2, x_3, s_1 et sm_1.

On obtient ainsi le modèle ARB^r_{eq} restreint par rapport à la propriété P^1_{eq}. Ce modèle possède deux fonctions de transition, F^1_t et Fm^1_t, ainsi que deux fonctions de sortie, F^1_o et Fm^1_o, étant bien plus petit que le modèle obtenu sur la base des dépendances structurelles. L'état initial de ARB^r_{eq} est calculé par rapport à l'état initial de ARB_{eq} ainsi qu'aux variables d'état présentes dans le cône : $\chi^r_0 = \bar{s}_1.\overline{sm}_1$.

Parcours symbolique

Compte-tenu du modèle réduit obtenu, la matrice de dépendance de ARB^r_{eq} a la forme suivante :

$$D_0 = \begin{array}{c} \\ F^1_t \\ Fm^1_t \\ \chi^r_0 \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & s_1 & sm_1 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Cette matrice est utilisable pour le premier pas du parcours symbolique. La fonction χ^r_0 se présente sous la forme d'un produit de deux termes à support disjoint. On peut donc envisager de partitionner la relation de transition de ARB^r_{eq}, avant d'aborder le calcul d'image. Par application de la formule (22), le premier pas du parcours se réécrit sous la forme :

$$\chi_1 = \exists x_1, x_2, x_3 : \big((\exists s_1 : (s'_1 \Leftrightarrow F^1_t) \,.\, \bar{s}_1) \,.\, (\exists sm_1 : (sm'_1 \Leftrightarrow Fm^1_t) \,.\, \overline{sm}_1)\big)$$

Les deux composantes de la relation de transition sont quantifiables séparément. On obtient :

$$\chi_1 = \exists x_1, x_2, x_3 : \big((s'_1 \Leftrightarrow x_1) \,.\, (sm'_1 \Leftrightarrow x_1)\big) = s'_1 \Leftrightarrow sm'_1$$

Le prédicat $o_1 = om_1$ est vrai dans le nouvel ensemble d'états atteint, car $\chi_1 \rightarrow (o_1 \Leftrightarrow om_1)$. Les états atteints jusqu'ici sont caractérisés par l'expression $\text{atteint}_1 = s_1 \Leftrightarrow sm_1$.

Dans la suite du parcours symbolique, le prédicat χ_1 ne peut plus être décomposé sous la forme d'un produit dont les termes ont un support disjoint. La matrice de dépendance D_1 reste identique à D_0, mais la partition de la relation de transition n'est plus envisageable.

On effectue donc une tentative pour rendre D_1 plus creuse, en appliquant une étape de décomposition disjonctive selon l'une des variables ayant un poids important dans la matrice de dépendance : la variable d'entrée x_1. Cette décomposition donne le résultat suivant :

$$\begin{aligned}\chi_2 = \exists x_1, x_2, x_3, s_1, sm_1 : (sm'_1 \Leftrightarrow Fm^1_t).(s'_1 \Leftrightarrow F^1_t).\chi_1 = \\ \exists x_2, x_3, s_1, sm_1 : (sm'_1 \Leftrightarrow Fm^1_t).(s'_1 \Leftrightarrow F^1_t)\big|_{x_1}.\chi_1\big|_{x_1} + \\ \exists x_2, x_3, s_1, sm_1 : (sm'_1 \Leftrightarrow Fm^1_t).(s'_1 \Leftrightarrow F^1_t)\big|_{\bar{x}_1}.\chi_1\big|_{\bar{x}_1}\end{aligned}$$

Examinons maintenant les matrices de dépendance D^+_1 (co-facteur $x_1 = 1$) et D^-_1 (co-facteur $x_1 = 0$) pour les deux termes obtenus ci-dessus :

$$D^+_1 = \begin{array}{l} \\ F^1_t \\ Fm^1_t \\ \text{atteint}_1 \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & s_1 & sm_1 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

$$D^-_1 = \begin{array}{l} \\ F^1_t \\ Fm^1_t \\ \text{atteint}_1 \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & s_1 & sm_1 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

La matrice D^+_1 suggère un ordre de quantification "au plus tôt" : $\{sm_1\}$, $\{s_1\}$, $\{x_2, x_3\}$. Le premier terme de la décomposition disjonctive est donné par l'expression : $sm'_1 \Leftrightarrow s'_1$. La matrice D^-_1 devient beaucoup plus simple : les fonctions de transition ne dépendent plus d'aucune variable. On obtient directement (sans quantification) le deuxième terme de la décomposition disjonctive : $\overline{sm}'_1.\bar{s}'_1$. En recomposant les deux termes obtenus, on obtient :

$\chi_2 = \overline{sm}'_1.\bar{s}'_1 + sm'_1 \Leftrightarrow s'_1 = sm'_1 \Leftrightarrow s'_1$.

Le prédicat $o_1 = om_1$ est vrai dans ce nouvel ensemble d'états atteint, car $\chi_2 \rightarrow (o_1 \Leftrightarrow om_1)$ Les états atteints lors de ce pas sont $\text{atteint}_2 = \text{atteint}_1 + \chi_2 = \text{atteint}_1$. C'est ici que le point fixe a été atteint. La propriété P^1_{eq} est vraie.

2.4.3.5 Moniteurs : preuve d'invariants

Sous contrainte de comportement passif du moniteur par rapport au modèle à vérifier (comme le rappellent les trois règles énoncées au paragraphe §2.3.3), la preuve des propriétés spécifiées par un moniteur procède elle aussi en trois étapes :

1. construction de la composition parallèle $M = M_{spec}||M_{impl}$, tout en respectant les contraintes de passivité relatives à M_{spec} (Figure 2.8) ;
2. pour chaque condition d'erreur c_{err} exprimée au sein de M_{spec}, prouver la propriété CTL $AG(\bar{c}_{err})$;

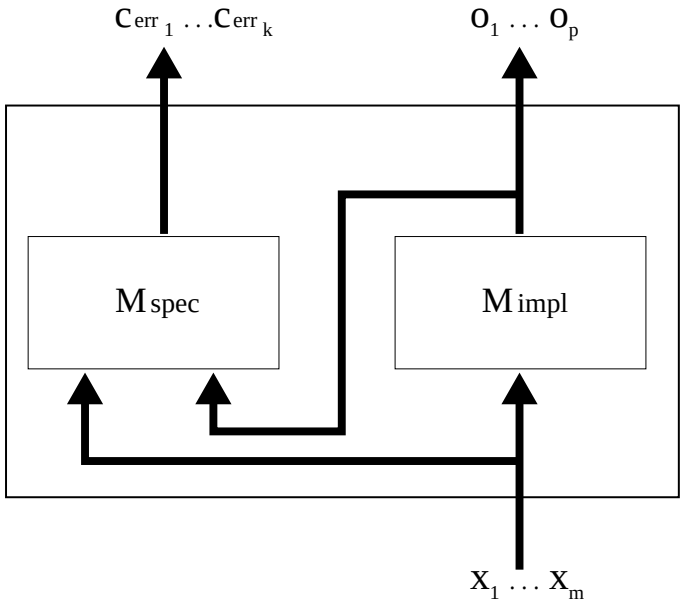

FIG. 2.8 – Produit autorisé entre un moniteur M_{spec} et une implémentation M_{impl}

La preuve par invariant en présence d'un moniteur hérite des indications d'optimisation du parcours symbolique applicables dans le cas de la preuve d'équivalence séquentielle, à la seule différence que la matrice de dépendance de M se rapproche d'une distribution triangulaire au lieu d'une distribution bloc diagonale.

2.4.3.6 Exemple : spécification d'un moniteur et preuve d'invariants

Une spécification opérationnelle tirant le meilleur parti de l'emploi d'un moniteur serait : *il est toujours vrai que si le canal 1 reçoit une requête ($x_1 = 1$) mais qu'il ne l'acquitte pas ($o_1 = 0$) alors une requête a été précédemment reçue sur le canal 1.*

Cette spécification fait référence au passé et donc n'a pas d'expression équivalente en CTL ou en LTL. Elle est exprimée grâce au moniteur M_{spec} suivant :

- ensemble des variables d'entrée : $X_{spec} = \{x_1, o_1\}$;
- ensemble des variables de sortie : $O_{spec} = \{P\}$;
- ensemble des variables d'état : $S_{spec} = \{sx\}$; cette unique variable d'état sert à mémoriser la valeur courante de x_1 ;
- fonction de transition : $sx' = Fs_t = x_1$;
- fonction de sortie : $P = Fs_o = (x_1 \ . \ \bar{o}_1) \rightarrow sx$; cette fonction exprime le fait que si l'entrée x_1 est activée sans être immédiatement acquittée, cette même entrée a dû être active dans l'état précédent ;
- état initial : $\chi_0^s = \overline{sx}$.

Preuve par évaluation d'invariant. La preuve de l'invariant P se fait sur le modèle composé $M_p = M_{spec}||ARB'$, en effectuant un parcours symbolique en avant. Pendant ce

parcours, on vérifie que P est vrai dans chaque état atteint de M_p. Le cône d'influence relatif à P contient les variables x_1, x_2, x_3, s_1, sx.

Le parcours se fait en quatre itérations. Le prédicat P est vrai dans tous les états de M_p.

2.4.4 Prise en compte des spécifications d'environnement

Une façon systématique d'intégrer les spécifications d'environnement est de les transformer en spécifications opérationnelles, à composer avec le système à vérifier.

Cas des spécifications logiques linéaires. Hormis pour le cas de la formule linéaire $GF\ f$ qui est traitée comme une contrainte d'équité, une façon systématique d'intégrer une spécification linéaire d'environnement $SPEC$ est de construire le tableau M_{SPEC} qui la satisfait et de composer ce tableau avec le modèle M à vérifier. On obtient ainsi le modèle $M_p = M_{SPEC}||M$. Les variables d'entrée de M dont le comportement est contraint par $SPEC$ seront des variables d'état dans le tableau M_{SPEC}. Les éventuelles contraintes d'équité nécessaires à ce tableau sont héritées au niveau de M_p.

Cas des spécifications opérationnelles Ecrire une spécification opérationnelle d'environnement revient à écrire un modèle M_{env} qui est composé avec le modèle à vérifier M. M_{env} peut lire à la fois les variables d'entrées et de sortie de M, mais doit écrire au moins une de ses variables d'entrée. On obtient ainsi le modèle $M_p = M_{env}||M$.

Si f est une formule temporelle quelconque, alors prouver que $M \models f$ sous une certaine contrainte d'environnement revient à prouver que $M_p \models f$.

2.5 Discussion

On distingue trois critères importants de comparaison des techniques de spécification formelle : la facilité d'écriture, le pouvoir d'expression et l'efficacité.

Les propriétés les plus faciles à écrire et à concevoir sont les invariants. Viennent ensuite les propriétés temporelles linéaires LTL, le langage FQL, ainsi que le sous-ensemble arborescent $\forall-$CTL. Les formules CTL sont plus difficiles à exprimer, en raison de la présence d'un ensemble d'opérateurs temporels plus nombreux, qui peuvent être facilement combinés. Le macro-langage SUGAR permet de rendre l'écriture de CTL plus aisée. Nous mettrons en dernière place la logique CTL*. Quant aux spécifications opérationnelles, elles sont bien souvent beaucoup plus lisibles et faciles à maintenir, lorsque leur taille reste extrêmement réduite.

Le classement selon le critère du pouvoir d'expression s'identifie au précédent classement renversé. Les invariants ont le moindre pouvoir expressif, alors que la logique CTL* est la plus puissante parmi celles qui ont été rappelées. Les spécifications opérationnelles restent bien sûr très flexibles.

Sur le plan de l'efficacité, ce sont les invariants qui sont les plus faciles à prouver, grâce à un parcours en avant, borné par l'ensemble des états atteignables du modèle vérifié. C'est

pourquoi il est essentiel de tenter de ramener la plupart des propriétés à vérifier à des invariants. Les moniteurs constituent un moyen très utile pour exprimer une propriété de sûreté d'un système à l'aide d'un invariant. Par ailleurs, les formules temporelles linéaires qui engendrent des tableaux terminaux peuvent elles-aussi être évaluées selon cette approche, car la formule CTL $\overline{EF\ \chi_{C_{eq}}}$ peut se réécrire sous forme d'un invariant : $AG\ \bar{\chi}_{C_{eq}}$.

Dans l'impossibilité de se ramener à une preuve d'invariant, la seule alternative d'évaluation restante est le parcours symbolique en arrière. En règle générale, ce parcours démarre avec l'ensemble des états potentiels du système vérifié. Ceci signifie que le nombre d'itérations nécessaires pour atteindre un point fixe peut être très élevé sur des systèmes relativement complexes (plus de 300 variables d'état). Pour palier ce problème, un calcul préalable de l'ensemble des états atteignables du système se montre souvent salutaire. Il permet d'une part de limiter la taille des termes calculés durant le parcours symbolique, et d'autre part de borner dans beaucoup de cas le nombre d'itérations de point fixe. L'application de cette heuristique nécessite toutefois des précautions. Ainsi, lorsque le système analysé contient, par exemple, un compteur sur 20 bits, l'analyse d'atteignabilité de cette construction est extrêmement coûteuse, car elle nécessite 10^6 itérations. C'est pourquoi, il est important de restreindre, lorsqu'il est possible, la taille des compteurs contenus dans une description. A défaut de cela, le parcours en arrière non contraint par l'ensemble d'états atteignables peut se révéler plus efficace.

Dans les exemples que nous avons analysés, nous avons montré l'importance de la matrice de dépendance pour l'efficacité du parcours symbolique, surtout dans le cas de la preuve d'équivalence séquentielle. Compte-tenu de son contenu, cette matrice est relativement facile à stocker et à analyser.

En cas d'échec de la preuve d'équivalence séquentielle dû à la complexité de la preuve, l'alternative implémentée dans de nombreux outils industriels de vérification est celle de la correspondance des variables d'état (lorsqu'il est possible de l'établir). La preuve peut ainsi être effectuée de manière compositionelle, comme dans le cas du raffinement. En revanche, en cas de non-équivalence, le diagnostic reste très difficile.

L'ajout des spécifications d'environnement tend généralement à rendre le modèle plus complexe, à travers l'opérateur de composition parallèle, et donc plus difficile à vérifier. Quelques cas très simples font exception à cette règle ; ils seront approfondis dans la suite de ce document.

Enfin, soulignons l'utilité du cône d'influence pour l'efficacité de la preuve. Prenons encore une fois l'exemple de l'arbitre, modélisé par ARB'. Cet exemple a été confié à un outil industriel de vérification. On a spécifié la propriété d'exclusion mutuelle : il n'y a jamais plus d'une, parmi les trois sorties de l'arbitre, qui est active à un instant donné. L'outil prouve que cette propriété est vraie. Il l'évalue comme un invariant et calcule l'ensemble des états atteignables de ARB', puis rend compte du nombre d'itérations nécessaires pour atteindre cet ensemble. Cependant, le prédicat exprimant l'exclusion mutuelle $EM = \overline{(o_1.o_2 + o_1.o_3 + o_2.o_3)}$ est une tautologie ! Le calcul du cône d'influence optimisé (Algorithme 2.3) de ARB' par rapport à EM trouve un cône vide, car le BDD de EM est trivial. La preuve de la formule $AG\ (EM = 1)$ par l'approche de l'invariant s'arrête à la simple construction d'un BDD pour représenter EM.

Il faut tout de même remarquer que la formule temporelle CTL $AG\ 1$ a aussi une

interprétation non triviale. Même si le prédicat $M \models AG\ 1$ est trivialement vrai, la formule CTL $AG\ 1$ identifie l'ensemble des états atteignables d'un modèle. Dans certains outils de vérification de modèles, tels que VIS [45], le fait de spécifier cette formule indique à l'outil que l'on souhaite calculer cet ensemble, sans aucune restriction sur le cône d'influence de M.

En dehors du cas où la réponse est triviale, l'emploi du cône optimisé donne des résultats intéressants même sur l'exemple très réduit de l'arbitre.

Deuxième partie

Stratégies de Vérification

Chapitre 3

Stratégies de preuve

3.1 Le besoin de stratégies

L'emploi des algorithmes de vérification symbolique de modèles se heurte rapidement aux limites de complexité exponentielle (en taille de représentation) inhérentes à cette technique. L'ensemble des états d'un modèle est représenté de manière implicite, grâce aux fonctions caractéristiques, ce qui améliore considérablement ses capacités. En contre-partie, la représentation par BDDs de ces fonctions est elle, limitée par une complexité spatiale asymptotique exponentielle en nombre de variables.

De nombreux travaux de recherche tentent de repousser ces limites. Chacune parmi les différentes approches proposées agit sur un point précis dans le flot de vérification proposant une optimisation. Ainsi, les modèles booléens béneficient de nombreuses techniques de minimisation logique très efficaces. La construction d'un modèle symbolique s'appuie sur des paquetages extrêmement performants, permettant de construire et de gérer des BDDs avec une utilisation optimale de la mémoire. Ces paquetages offrent également des algorithmes pointus et efficaces pour effectuer les opérations booléenes de base, nécessaires lors du parcours symbolique. Des techniques automatiques permettent de limiter la taille du modèle symbolique pour la preuve, en tentant une décomposition structurelle basée soit sur le graphe de dépendance, soit sur les propriétés de symétrie du modèle. Enfin, de nombreuses heuristiques permettent d'optimiser tant le calcul des états atteignables d'un modèle que la preuve des propriétés temporelles.

L'ensemble de ces techniques améliorent considérablement les performances de la vérification symbolique de modèles. Cependant, cela n'implique qu'une augmentation limitée de la capacité de cette technique ; la taille des modèles industriels réellement vérifiables reste bien en dessous des besoins.

Cette situation a déterminé, au sein de la communauté des utilisateurs, le franchissement volontaire des frontières à l'intérieur desquelles la vérification de modèles s'effectue de manière automatique. Ainsi, l'utilisateur[1] apporte sa connaissance a priori sur la structure et/ou le comportement du modèle qu'il souhaite vérifier. En s'appuyant sur cette connaissance, il met en place une *stratégie* permettant de guider l'outil de vérification dans le

[1] en occurence, soit le concepteur, soit l'ingénieur de vérification

but d'éviter l'explosion combinatoire. Toute stratégie de vérification s'appuie sur les points suivants :

1. le choix de la technique de spécification et de preuve à utiliser :
   - choix de la méthode de spécification : logique (linéaire, arborescente), ou opérationnelle ;
   - pour une méthode de spécification choisie, déterminer quel est l'algorithme de preuve le plus adapté ainsi que la représentation susceptible de donner les meilleurs résutats. Actuellement le choix peut se porter soit sur la vérification symbolique basée sur des BDDs ou sur les techniques SAT, soit sur la technique de vérification de modèles bornés BMC ;
   - pour une technique de preuve choisie, déterminer quel genre de parcours, en avant ou en arrière, serait plus adapté. En cas d'un parcours en arrière, déterminer si un calcul préalable de l'ensemble des états atteignables peut ou non être bénéfique, etc. ;
2. le choix de la façon d'appliquer une technique de preuve sur le modèle à vérifier. Pour une spécification donnée :
   - on identifie les parties du modèle qui sont seules concernées par la preuve ;
   - on identifie des buts de preuve intermédiaires qui, une fois prouvés, permettent soit de déduire la validité de la spécification initiale, soit de conclure à un taux de couverture fonctionnelle satisfaisant, dans le cas où une réponse formelle ne peut être obtenue (à cause de la complexité du modèle).

Alors que la première étape ci-dessus se résume à un choix d'outils, la deuxième permet de spécifier quelle est la manière la plus efficace d'utiliser ces outils ; elle sous-entend une intéraction importante avec l'utilisateur. L'approche la plus efficace est basée sur la méthode "diviser pour règner" : on décompose la tâche de vérification selon deux directions orthogonales : structurelle et comportementale.

L'approche structurelle a pour but de décomposer un modèle en un ensemble de parties distinctes. Lorsqu'il s'agit d'un modèle hiérarchique, il contient un ensemble de composants qui sont eux mêmes des modèles hiérarchiques. Soit C l'ensemble de tous les nœuds (composants) présents dans une hiérarchie de modèles. Par application de cette approche, on vérifie tous les modèles correspondant aux éléments de l'ensemble 2^C.

Si le modèle ne comporte aucune indication de hiérarchie, la décomposition ne peut se faire qu'en isolant l'une après l'autre les parties du modèle qui implémentent des fonctionnalités précises. Pour vérifier un tel modèle, on vérifie séparément les différentes parties isolées. Cela nécessite des connaissances approfondies au niveau de l'implémentation.

L'approche comportementale permet de visualiser un modèle comme un ensemble de fonctionnalités indépendantes. Pour vérifier le modèle, on peut se focaliser successivement sur chacune des fonctionnalités existantes. Ainsi, il est parfois possible de scinder une formule temporelle en plusieurs sous-formules plus simples, chacune pouvant être prouvée séparément. Chaque sous-formule concerne une fonctionnalité distincte. La même approche peut être appliquée au niveau de l'environnement. Les différents comportements possibles dans l'environnement sont partagés en scénarios distincts. Le même modèle est successivement composé et vérifié avec chacun des scénarios décrits. Contrairement au découpage

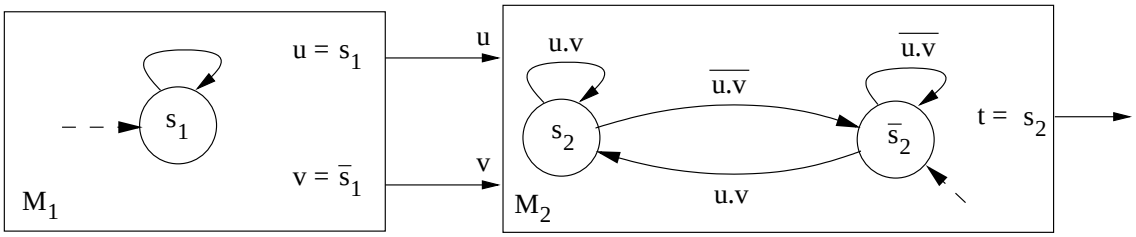

FIG. 3.1 – Effet de la composition de deux modèles sur la validité de la preuve : $M_2 \models EX\ t$ mais $M_1||M_2 \not\models EX\ t$

structurel, le découpage comportemental se montre parfois plus commode, car il ne nécessite pas de connaissances approfondies sur la structure et les choix d'implémentation du modèle vérifié.

La suite de ce chapitre passe en revue les différentes stratégies de vérification existantes. Chaque méthode basée sur la décomposition (structurelle ou comportementale) est accompagnée du raisonnement permettant de déduire la validité d'une propriété à partir de celle de ses différentes composantes.

3.2 Approche structurelle

3.2.1 Méthode descendante

3.2.1.1 Décomposition simple

Cette démarche est très naturelle, car elle est calquée sur les étapes du flot de conception modulaire. Lorsque l'on suit ce flot de conception, on procède par rafinements successifs du document de spécification. Les fonctions complexes sont récursivement décomposées en modules à la fois simples et faciles à réutiliser. L'implémentation commence par les modules les plus simples, puis recombine ceux-ci en des modules de plus en plus complexes.

Le modèle résultant est souvent trop complexe pour permettre d'évaluer une propriété temporelle, même très simple. En revanche, compte-tenu du fait que chacun de ses composants a été spécifié séparément, il peut également être vérifié séparément. Une fois que chaque composant a été vérifié indépendamment des autres, se pose le problème de déduire les propriétés du modèle global.

Le schéma de raisonnement le plus simple permet de réunir au niveau global les propriétés temporelles vérifiées à l'échelle locale de chaque composant. Ainsi, si P et Q sont deux formules temporelles ayant été validées sur deux composants d'un modèle M, alors on souhaite pouvoir déduire que $M \models P.Q$. Cependant, l'application systématique de ce raisonnement n'a de validité que sur un sous-ensemble de la logique CTL. Pour illustrer ce fait, examinons l'exemple suivant :

Exemple. Soit M_1 et M_2 deux modèles symboliques, dont les graphes d'état respectifs sont reproduits dans la figure 3.1. Voici la description symbolique de ces deux modèles. Le

modèle M_1 :
- variables d'entrée : $X_1 = \emptyset$;
- variables de sortie : $O_1 = \{u, v\}$;
- variables d'état : $S_1 = \{s_1\}$;
- fonction de transition : $s_1' = 1$;
- fonctions de sortie :
  $u = s_1$;
  $v = \bar{s}_1$;
- état initial : $\chi_0^1 = s_1$;

Le modèle M_2 :
- variables d'entrée : $X_2 = \{u, v\}$;
- variables de sortie : $O_2 = \{t\}$;
- variables d'état : $S_2 = \{s_2\}$;
- fonction de transition : $s_2' = u.v$;
- fonction de sortie : $t = s_2$;
- état initial : $\chi_0^2 = \bar{s}_2$;

Le modèle M_1 satisfait trivialement la formule temporelle CTL $P_1 = AG(u = 1)$. Quant au modèle M_2, il est facile de vérifier qu'il satisfait la formule $P_2 = EX(t = 1)$.

Construisons le modèle $M = M_1||M_2$. On remarque à présent que M continue à satisfaire P_1, mais qu'il ne satisfait plus P_2.
□

Ainsi, pour le raisonnement compositionnel, la logique temporelle CTL ne constitue pas une *congruence* vis-à-vis de la composition parallèle : $M \models P_{CTL} \nrightarrow M||M' \models P_{CTL}$

Intuitivement, la validité d'une formule contenant des quantificateurs existentiels de chemin dépend fortement de l'environnement du modèle au moment où cette formule est évaluée. En particulier, en absence d'une spécification d'environnement, il peut exister une séquence d'états (finie ou non) satisfaisant une formule de chemin quelconque. Cependant, en présence d'un environnement plus contraignant (moins de comportements possibles), ce même chemin peut ne plus être accessible.

Par contraste, si une formule contient uniquement des quantificateurs universels de chemin et si le modèle M satisfait cette formule, alors il continuera à la satisfaire quel que soit le comportement de son environnement. Ainsi ce sont les logiques temporelles ∀-CTL et LTL qui possèdent des propriétés très intéressantes pour le raisonnement compositionnel. Pour des raisons de simplicité, on ne parlera que des formules ∀-CTL, mais toutes les affirmations restent valables, sauf indication contraire, pour les formules LTL.

La conservation ascendante

Soit M_i un composant du modèle hiérarchique M. Si P_i est une formule ∀-CTL et si $M_i \models P_i$ alors :

1. quel que soit le modèle M_j, $M_i||M_j \models P_i$;
2. le modèle hiérarchique M satisfait P_i : $M \models P_i$;
3. si P_j est également une formule ∀-CTL et si $M_i \models P_j$ alors $M_i \models P_i.P_j$.

D'une manière plus générale, si $P = AGp$, et $Q = AGq$ où p et q sont des formules CTL quelconques, le même raisonnement compositionnel ne peut s'appliquer, sauf dans le

cas particulier où P et Q sont satisfaites par le même modèle. Dans ce cas, si $M_i \models P$ et $M_i \models Q$ alors on peut déduire que $M_i \models AGp.q$. Cependant, le modèle hiérarchique M ne satisfait pas forcément $AGp.q$

Application à l'exemple de l'arbitre modulaire Certaines propriétés du modèle ARB_{mod} peuvent être évaluées à l'échelle de ses composants. Par exemple, chaque canal $i \in 1..3$ de l'arbitre devrait satisfaire la propriété : $P_i = AG(o_i \rightarrow x_i)$. Chaque propriété P_i est une formule ∀-CTL. En outre, compte-tenu des variables dont elle dépend, elle peut être évaluée à l'échelle locale du composant $CANAL_i$.

Si toutes ces propriétés sont vraies à l'échelle de leur composant correspondant, alors en vertu de la conservation ascendante, le raisonnement suivant s'applique :

si $CANAL_1 \models P_1$ alors $ARB_{mod} \models P_1$
si $CANAL_2 \models P_2$ alors $ARB_{mod} \models P_2$
si $CANAL_3 \models P_3$ alors $ARB_{mod} \models P_3$

On obtient ainsi que $ARB_{mod} = CANAL_1||CANAL_2||CANAL_3 \models P_1.P_2.P_3$. Par application de la règle de conservation ascendente, on a pu déduire la validité des propriétés P_i au niveau global de ARB_{mod}.

La règle de conservation peut également être utilisée dans le but de décomposer des formules en parties plus simples. Rappelons la propriété d'exclusion mutuelle de M_{arb} :

$EM = AG(\overline{o_1.o_2 + o_1.o_3 + o_2.o_3})$

Cette propriété peut se réécrire sous la forme :

$$\begin{aligned} EM &= AG((\overline{o_1.o_2}).(\overline{o_1.o_3}).(\overline{o_2.o_3})) \\ &= AG((\bar{o}_1 + \bar{o}_2).(\bar{o}_1 + \bar{o}_3).(\bar{o}_2 + \bar{o}_3)) \\ &= AG(\bar{o}_1 + \bar{o}_2).AG(\bar{o}_1 + \bar{o}_3).AG(\bar{o}_2 + \bar{o}_3) \end{aligned}$$

La formule EM se décompose donc en trois sous-formules :

$EM_{12} = AG(\bar{o}_1 + \bar{o}_2)$
$EM_{13} = AG(\bar{o}_1 + \bar{o}_3)$
$EM_{23} = AG(\bar{o}_2 + \bar{o}_3)$

Ainsi :

si $CANAL_1||CANAL_2 \models EM_{12}$ alors $ARB_{mod} \models EM_{12}$
si $CANAL_1||CANAL_3 \models EM_{13}$ alors $ARB_{mod} \models EM_{13}$
si $CANAL_2||CANAL_3 \models EM_{23}$ alors $ARB_{mod} \models EM_{23}$

En résumant, on peut déduire que $ARB_{mod} \models EM$. Grâce à cette démarche de décomposition, la propriété EM a pu être prouvée sans évalution à l'échelle globale de ARB_{mod}.

3.2.1.2 Raisonnement "Je suppose-tu garantis"

L'inconvénient majeur de la technique de décomposition simple vient du fait que la preuve d'un composant fait systématiquement abstraction de l'existence d'un environnement. Ainsi, si le composant M_i satisfait la propriété ∀-CTL P_i, alors le fait de composer M_i avec d'autres modules, qui déterminent ainsi son environnement réel, n'aura aucune influence sur la validité de P_i. Cependant, dans le cas d'un système réel, cette situation très favorable ne se produit que rarement. Il est en revanche bien plus courant de se retrouver devant un échec de la preuve $M_i \models P_i$. Cet échec peut avoir deux significations :

- à cause d'une erreur de conception, le composant M_i ne satisfait pas la propriété P_i ;
- le composant M_i a été conçu pour fonctionner dans un environnement adéquat, en absence duquel la correction de son fonctionnement n'est pas garantie. Le résultat obtenu est connu sous le nom de "faux négatif".

Avant toute tentative de preuve, il est donc indispensable de s'assurer que le composant possède un environnement adéquat. Soit M un modèle hiérarchique et M_i et M_j deux de ses composants tels que les variables d'entrée de M_i correspondent aux variables de sortie de M_j : $O_j = X_i$. Si M_i a été conçu en fonction d'un environnement précis, mis en œuvre par M_j, alors pour prouver que $M_i \models P_i$ en suivant une approche compositionnelle, il est nécesaire de tenir compte de la présence de M_j. Ainsi, il serait nécessaire de prouver que $M_j||M_i \models P_i$. Toutefois, le modèle résultant $M_j||M_i$ peut lui aussi avoir besoin d'un environnement adéquat. Cet environnement peut être obtenu en élargissant au fur et à mesure le modèle symbolique à d'autres composants de M. En suivant cette approche, lorsque l'environnement obtenu est enfin satisfaisant, le modèle symbolique composé qui doit être utilisé pour la preuve de P_i peut être bien plus complexe que M_i, au détriment de l'efficacité de la décomposition.

La preuve $M_i \models P_i$ nécessite seulement un ensemble d'hypothèses portant sur la correction de l'environnement de M_i. Ces hypothèses de correction d'environnement peuvent souvent être formellement spécifiés grâce à un ensemble de formules temporelles. Soit F_{ji} un ensemble de formules temporelles décrivant quelques comportements de M_j, pouvant influencer le fonctionnement de M_i. F_{ji} consitue une spécification d'environnement pour le composant M_i. En même temps, F_{ji} constitue une spécification du composant M_j. Ainsi, la preuve $M_i \models P_i$ suit le déroulement suivant :

1. prouver que
$$\bigwedge_{f \in F_{ji}} f \rightarrow (M_i \models P_i)$$
2. prouver que $\forall f \in F_{ji}, M_j \models f$.

Le parcours de ces deux étapes permet de déduire que (1) $M_i \models P_i$ compte-tenu des hypothèses représentées par F_{ji}, et que (2) ces hypothèses sont effectivement satisfaites par M_j. Ceci certifie le fait que tous les comportements supposés dans l'environement de M_i font partie de M_j, et donc que la vérification de M_i s'est appuyé uniquement sur des hypothèses "réalistes". Ceci implique que $M_j||M_i \models P_i$. C'est à partir de cet instant que l'on peut appliquer la règle de conservation afin de conclure que P_i est vraie au niveau global du modèle hiérarchique M.

Le schéma de raisonnement ci-dessus est récursif. Lors de la deuxième étape, la preuve $M_j \models f$ peut faire appel au même procédé, appliqué sur l'environnement de M_j. Cet enchaînement ne s'arrête que lorsque toutes les preuves $M_j \models f$ aboutissent quel que soit l'environnement de M_j.

Ce procédé de décomposition basé sur l'alternation hypothèse/preuve est connu sous le nom de *raisonnement "je suppose-tu garantis"*[2] [73].

[2] de l'anglais : *Assume-Guarantee*

Remarque. Il est clair que dans ce genre de raisonnement, la profondeur de récursivité doit être finie. Par conséquent, cette approche n'est pas applicable dans le cas où le raisonnement devient circulaire : si P_i est une propriété de M_i et P_j une propriété de M_j, alors prouver que

$P_j \rightarrow M_i \models P_i$ puis

$P_i \rightarrow M_j \models P_j$

n'entraîne aucun résultat concernant la validité de P_i et de P_j à l'échelle globale du modèle hiérarchique M qui contient M_i et M_j. Dans ce cas de figure, on est obligé de prouver que $M_i||M_j \models P_i$ et que $M_i||M_j \models P_j$. D'une manière plus générale, la nécessité d'hypothèses d'environnement pour la preuve d'un composant engendre une relation de dépendance entre les composants d'un modèle hiérarchique. Cette relation n'est pas reflexive mais possède une propriété de transitivité structurelle : si l'environnement de M_i est engendré par M_j et si l'environnement de M_j est engendré par M_k, alors le comportement de M_k *peut* avoir une influence sur le comportement de M_i. Ainsi, ces composants peuvent constituer les nœuds d'un graphe de dépendance. Si les nœuds de ce graphe peuvent être triés topologiquement, alors un raisonnement "je suppose-tu garantis" est applicable. Dans le cas contraire, le graphe contient des cycles, ce qui engendrerait un raisonnement cyclique. Pour palier ce problème, il est nécesaire de construire un tri topologique sur les composantes strictement connexes du graphe de dépendance.

Une fois de plus et pour les même raisons citées au paragraphe précédent, ce sont les logiques temporelles qui n'incluent pas d'opérateurs existentiels de chemins, c'est à dire $\forall - CTL$ ou bien LTL, qui confèrent robustesse à ce genre de raisonnement compositionnel.

Dans le cas où le modèle M est spécifié exclusivement à l'aide de la logique temporelle linéaire LTL, le raisonnement compositionnel s'adapte de la façon suivante :

1. prouver que
$$\bigwedge_{f_{LTL} \in F_{ji}} f_{LTL} \rightarrow (M_i \models P^i_{LTL})$$
L'hypothèse d'environnement f_{LTL} peut être traduite en une spécification opérationnelle (tableau) $M_{f_{LTL}}$ accompagné, si nécessaire, d'un ensemble de contraintes d'équité. Le tableau obtenu représente exactement les traces qui satisfont f_{LTL}. Donc, cela revient à restreindre M_i aux seuls comportements acceptés par $M_{f_{LTL}}$. Ensuite, on évalue $M_{f_{LTL}}||M_i \models P^i_{LTL}$ grâce à un des algorithmes de preuve LTL existants ;
2. prouver que $\forall f_{LTL} \in F_{ji}, M_j \models f_{LTL}$. Cette étape se réalise tout simplement par construction successive des tableaux $M_{h_{LTL}}$ où $h_{LTL} = \bar{f}_{LTL}$ dans le but d'appliquer encore une fois un algorithme de preuve LTL.

La performance de cette approche basée sur la logique linéaire reste bien sûr limitée par la performance inhérente à l'évaluation d'une formule LTL, performance qui dépend également de la *force* des tableaux générés.

Le cas où le modèle M est spécifié exclusivement à l'aide de $\forall$-CTL est bien plus intéressant. En effet, toute formule $\forall$-CTL f possède également un tableau correspondant M_f, tel que $M_f \models f$ et si M_i est un modèle quelconque, alors on a : $M_i \models f \Leftrightarrow M_i \preceq_s M_f$. Ainsi, le critère de satisfaction d'une formule $\forall$-CTL est synonyme de simulation entre le

1	$\forall M_i, M_j : M_i \| M_j \preceq_s M_i$
2	$\forall M_i, M_j, M_k : M_i \preceq_s M_j \rightarrow M_i \| M_k \preceq_s M_j \| M_k$

TAB. 3.1 – Lemmes utiles pour l'appui du raisonnement compositionnel

modèle à vérifier et le tableau de f. Ce résultat a été présenté dans [46, 21]. La mise en œuvre de ce raisonnement est similaire au cas LTL : on utilise des tableaux en tant que spécifications opérationnelles d'environnement. La preuve peut se faire, au choix, soit grâce aux algorithmes de preuve CTL, soit par preuve de simulation, à travers la construction d'un tableau.

Le raisonnement "je suppose-tu garantis" peut ainsi être centré autour de la relation de simulation. Dans ce cadre, un raisonnement quelconque peut être justifié grâce notamment aux lemmes énumérées dans le tableau 3.1.

Dans de nombreux cas cependant, la preuve $M_i \models P_i$ où P_i est une formule $\forall$-CTL peut s'avérer bien plus efficace que la preuve de simulation $M_i \preceq_s M_{P_i}$, dont la complexité dépend à la fois de la taille de M_i et de celle de M_{P_i}.

Dans le paragraphe suivant, nous illustrons l'application de la méthode "je suppose-tu garantis" à la vérification de propriétés $\forall$-CTL de l'arbitre modulaire ARB_{mod}.

Exemple : application à la vérification de l'arbitre modulaire. Une propriété très intéressante de ARB_{mod} exprime la vivacité de chaque canal. Pour $i = 1..3$ on a :

$CANAL_i \models AG(x_i \rightarrow AFo_i)$

Grâce à la méthode de raisonnement "je suppose-tu garantis" nous allons tenter de vérifier que $ARB_{mod} \models AG(x_i \rightarrow AFo_i)$ pour $i = 1..3$.

On remarque que l'arbitre ne satisfait pas ces propriétés lorsqu'il est placé dans un environnement quelconque. En effet, une fois activée, chacune des requêtes peut se retracter à tout moment. L'arbitre ne mémorise pas l'état de ses requêtes. Si une requête devient inactive avant d'avoir été acquitée, alors par construction de l'arbitre elle ne sera jamais acquitée. Si un tel scénario se produit, la propriété de vivacité de ARB_{mod} est violée.

Il est raisonnable de considérer que les seuls comportements admissibles dans l'environnement de ARB_{mod} sont ceux où, une fois activée, la requête x_i reste active jusqu'à ce que o_i devienne actif, après quoi elle est désactivée. Ce comportement peut se reproduire indéfiniment. La figure 3.2 représente un graphe d'états très simple d'une spécification opérationnelle d'environnement ENV^i_{arb} qui met en œuvre ce mécanisme de persistence. La requête x_i est activée lorsque ENV^i_{arb} se trouve dans l'état étiqueté e. Le modèle ENV^i_{arb} peut effectuer une transition non-déterministe, modélisée par la pseudo-entrée ND. Cela signifie que la requête x_i peut être activée à tout moment. En revanche, lorsque $o_i = 1$, la requête x_i sera désactivée car ENV^i_{arb} retourne dans l'état étiqueté $\bar{e}$. Il est facile de constater que ENV^i_{arb} satisfait trivialement la propriété de persistence attendue : une fois la requête activée, elle restera active jusqu'à ce qu'elle soit acquittée.

La preuve $CANAL_1 \models AG(x_1 \rightarrow AFo_1)$

Cette preuve se fera sous l'hypothèse de requête d'environnement persistente. Ainsi, le modèle symblique pour la preuve est constitué de $CANAL_1 \| ENV^1_{arb}$. Par construction

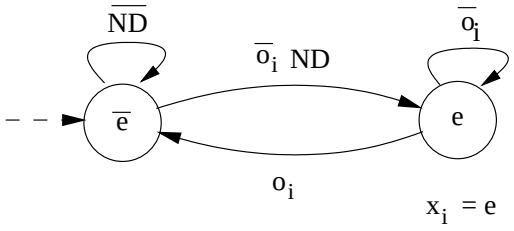

FIG. 3.2 – Modèle exprimant l'environnement de $CANAL_i$

de ARB_{mod} (figure 2.7 page 66), la variable d'entrée $prio_{e1}$ du composant $CANAL_1$ a la valeur constante 0.

L'évaluation symbolique permet d'établir que la propriété $CANAL_1||ENV^1_{arb} \models AG(x_1 \rightarrow AFo_1)$ est satisfaite. Ainsi, on peut conclure qu'en présence de la même hypothèse de persistance on a : $ENV^1_{arb}||ARB_{mod} \models AG(x_1 \rightarrow AFo_1)$.

La preuve $CANAL_2 \models AG(x_2 \rightarrow AFo_2)$
Pour commencer, on va considérer que cette preuve bénéficie des mêmes hypothèses d'environnement que précédemment. On travaille cette fois-ci sur le modèle symbolique $CANAL_2||ENV^2_{arb}$. Cependant, contrairement au cas de $CANAL_1$, nous ne pouvons plus supposer que l'entrée $prio_{e2} = 0$, car ceci serait à priori contraire à la réalité. On va supposer que $prio_{e2}$ peut avoir un comportement quelconque.

Une première tentative de prouver que $CANAL_2||ENV^2_{arb} \models AG(x_2 \rightarrow AFo_2)$ échoue. N'importe quel outil de vérification peut construire un contre-exemple contredisant cette propriété. En effet, compte tenu de l'hypothèse sur le comportement de $prio_{e2}$, il est possible de construire une séquence de valeurs d'entrée de longueur infinie selon laquelle à partir d'un certain moment $prio_{e2} = 1$. Une telle séquence empêche $CANAL_2$ d'acquiter ses requêtes entrantes.

Nous nous trouvons ainsi en présence d'un cas typique où la décomposition simple échoue. Les hypothèses d'environnement retenues pour $CANAL_2$ ne tiennent pas compte de la réalité : au sein de ARB_{mod}, le comportement de $prio_{e2}$ est défini par $CANAL_1$.

Cherchons à construire C_{12}, un ensemble de formules temporelles traduisant l'influence de $CANAL_1$ sur le comportement de $CANAL_2$. De manière plus intuitive, notre objectif est de spécifier une contrainte d'équité sur $prio_{e2}$, permettant de s'assurer que cette variable prendra la valeur 0 infiniment souvent. Pour commencer, il est utile de savoir si le scénario qui contredit $AG(x_2 \rightarrow AFo_2)$ peut être produit par $CANAL_1$: est-il vrai que $CANAL_1 \models EF\ EG prio_{s1}$? En évaluant cette formule, on trouve que la réponse est négative. Par conséquent, on a : $CANAL_1 \models AG\ AF\overline{prio}_{s1}$. Nous allons retenir cette propriété comme hypothèse d'environnement pour la vérification de $CANAL_2$: $C_{12} = \{AG\ AF\overline{prio}_{s1}\}$.

La preuve $CANAL_2||ENV^2_{arb} \models AG(x_2 \rightarrow AFo_2)$ sous l'hypothèse que tous les éléments de C_{12} sont vrais donne cette fois-ci une réponse positive.

Pour résumer, on a prouvé que :

1. $(\bigwedge_{f \in C_{12}} f) \rightarrow (CANAL_2||ENV^2_{arb} \models AG(x_2 \rightarrow AFo_2))$;
2. $\forall f \in C_{12} : CANAL_1 \models f$;

Par conséquent, $ENV^1_{arb}||ENV^2_{ARB}||CANAL_1||CANAL_2 \models AG(x_2 \rightarrow AFo_2)$. En appliquant la règle de conservation on déduit que :

$$ENV^1_{arb}||ENV^2_{ARB}||ARB_{mod} \models AG(x_2 \rightarrow AFo_2).$$

La preuve $CANAL_3 \models AG(x_3 \rightarrow AFo_3)$
En suivant la même démarche que précedemment, on construit C_{23}, l'ensemble des formules temporelles qui traduisent l'influence de $CANAL_2$ sur le comportement de $CANAL_3$. A priori, on a $C_{23} = \{AG\ AF\overline{prio}_{s2}\}$. L'application des étapes du raisonnement compositionnel donne les résultats suivants :

1. $(\bigwedge_{f \in C_{23}} f) \rightarrow (CANAL_3||ENV^3_{arb} \models AG(x_2 \rightarrow AFo_2))$. Cette propriété est vraie sous l'hypothèse f ;
2. $\forall f \in C_{23} : CANAL_2 \models f$. Cette preuve se solde par un échec. Vraisemblablement $CANAL_2$ a besoin de quelques hypothèses d'environnement supplémentaires. On va donc tenter de prouver que $CANAL_2 \models f$ sous l'hypothèse C_{12}. Cette nouvelle tentative se solde par un nouvel échec. Le contre-exemple exhibe un scénario selon lequel f n'est jamais vraie, car lorsque $CANAL_2$ est connecté à $CANAL_1$, il peut arriver que le prédicat $prio_{s2}$ soit toujours vrai. En réalité, si $CANAL_1$ et $CANAL_2$ ont des requêtes trop fréquentes, $CANAL_3$ peut se retrouver bloqué.

L'échec de cette troisième étape de preuve est synonyme d'*absence de réponse*. En effet, cet échec peut avoir deux causes distinctes :

- le composant $CANAL_3$ est erroné (ce qui dans notre cas est peu vraisemblable) ;
- les hypothèses d'environnement C_{12} et C_{23} sont insuffisantes : il est peut-être nécessaire de les enrichir avec de nouvelles formules temporelles, permettant de modéliser l'environnement de manière plus précise ;
- la propriété que l'on souhaite prouver est exprimée de manière incorrecte.

Compte-tenu de la taille réduite de ARB_{mod} nous pouvons tenter de prouver que :
$ENV^1_{arb}||ENV^2_{arb}||ENV^3_{ARB}||ARB_{mod} \models AG(x_3 \rightarrow AFo_3)$.
C'est cette preuve qui va nous confirmer que la formule $AG(x_3 \rightarrow AFo_3)$ n'est pas satisfaite.

Par conséquent, lorsque le raisonnement "je suppose-tu garantis" échoue, il est difficile de discerner quelle est la véritable cause de cet échec. Si toutefois nous avons quand même pu obtenir une réponse, c'est grâce à la taille réduite du modèle vérifié.
□

3.2.1.3 Décomposition de la preuve de raffinement

La décomposition hiérarchique s'applique avec succès dans le cas de l'évaluation de specifications logiques. Cependant, lorsque l'on choisit l'écriture d'un modèle abstrait à la place d'une formule temporelle, on est également confronté avec la difficulté de preuve de la simulation ou du raffinement, causée par la complexité des deux modèles comparés. Une démarche compositionnelle similaire à la méthode "je suppose - tu garantis" serait très utile. Si l'implémentation et le modèle abstrait ont la même structure hiérarchique, on souhaite pouvoir comparer successivement un composant de l'implémentation avec son correspondant du modèle abstrait, et d'en déduire des propriétés au niveau global.

Le raisonnement compositionnel basé sur la relation de simulation constitue le fondement d'une approche de spécification et de preuve proposée dans [50]. Pour palier le problème de complexité typique de la preuve de simulation, les auteurs proposent d'utiliser sa variante restreinte : le raffinement ($\preceq_r$). Comme il a été rappelé au chapitre précédent, la relation de raffinement ne peut être établie que lorsque les variables d'état du modèle abstrait peuvent être mises en correspondance avec des variables d'état de l'implémentation. Ainsi, cette approche requiert une intervention importante de la part de l'utilisateur, dans le but d'enrichir à la fois le modèle abstrait et l'implémentation jusqu'à ce que cette correspondance soit atteinte [49]. Si le modèle abstrait M_{abs} contient une variable d'état v_{abs} qui n'est pas présente dans l'implémentation M_{impl}, alors il est nécessaire d'enrichir l'implémentation afin que les critères suivants soient satisfaits :

- $S'_{impl} = S_{impl} \cup v_{abs}$. La variable v_{abs} doit faire partie de l'implémentation ;
- la fonction de transition de v_{abs} doit expliciter une correspondance entre les états de M_{abs} et ceux de M_{impl} ;
- aucune fonction de transition de $v \in S_{impl}$ ne dépend de v_{abs} ;

Cette description complémentaire, qui enrichit une implémentation afin de la rapprocher de son modèle abstrait, est connue sous le nom de *témoin*[3]. Dans le cadre d'une démarche basée sur la preuve de raffinement, l'écriture d'un témoin se fait manuellement. Les auteurs affirment également que $M_{impl} \preceq_s M_{abs}$ si et seulement s'il existe un témoin T_{abs} contenant les variables d'état de M_{abs} qui ne font pas partie de M_{impl}, tel que $M_{impl}||T_{abs} \preceq_r M_{abs}$.

La démarche de décomposition proposée par cette approche s'apparente elle aussi au raisonnement "je suppose-tu garantis". Mais elle est toutefois différente de celle que nous avons examiné précédemment : on tente de prouver que $M_{impl} \preceq_r M_{abs}$, où $M_{impl} = M^1_{impl}||...||M^m_{impl}$ et $M_{abs} = M^1_{abs}||...||M^n_{abs}$. Cette preuve se décompose de la façon suivante.

Soient $G_i, i = 1..n$ des compositions de modèles arbitrairement choisis parmi les composants de M_{abs} et de M_{impl}, à condition qu'ils soient deux à deux composables et que G_i ne contienne pas M^i_{abs}. Si

$$\forall i = 1..n : G_i \preceq_r M^i_{abs}$$

alors $M_{impl} \preceq_r M_{abs}$.

La construction de G_i doit préserver la correspondance des variables d'état et de sortie : chaque variable d'état de M^i_{abs} doit être une variable d'état de G_i (éventuellement après ajout d'un témoin) et chaque variable de sortie de M^i_{abs} doit être une variable de sortie d'exactement un composant de G_i. Cette approche de raisonnement a été implémentée dans l'outil de vérification MOCHA.

Cas de la correspondance totale entre variables d'état. Lorsque la correspondance des variables d'état de l'implémentation et du modèle abstrait est totale, la preuve de raffinement peut être mise en œuvre par comparaison des fonctions de transition comme le montre l'équation (23). Cette équation constitue seulement une réécriture de l'équation (21).

[3] de l'anglais : *Witness*

$$\chi_{ATT}(\langle s_n \rangle) \rightarrow \exists \langle ND_r \rangle : \prod_{i=1}^{n} (F^i_{impl}(\langle x_m \rangle, \langle s_n \rangle) \Leftrightarrow F^i_{abs}(\langle x_m \rangle, \langle ND_r \rangle, \langle s_n \rangle)) \quad (23)$$

Cette approche possède deux désavantages. Tout d'abord, le calcul de χ_{ATT} peut se révéler très coûteux, à cause de la taille de l'implémentation. Ensuite, se pose le problème de la représentation symbolique du produit des termes ($F^i_{impl} \leftrightarrow F^i_{abs}$), qui peut elle aussi être très coûteuse.

Montrons de quelle manière la technique "je suppose-tu garantis" est applicable dans cette situation. On observe tout d'abord que n'importe quel modèle booléen décrit par un ensemble de fonctions de transition peut être visualisé comme une composition parallèle de composants atomiques. Si M possède les fonctions de transition $F^i_t, 1 = 1..n$, les variables d'état courant $\langle s_n \rangle$ et prochain état $\langle s'_n \rangle$, alors on a :

$$\forall M : M = M_1 || \ldots M_n, \text{ tels que } \forall i = 1..n : X_i = X, O_i = O \text{ et } S_i = \{s_i\}$$

La relation de transition T_i de chacun de ces composants atomiques est de la forme :

$$T_i = (s'_i \Leftrightarrow F^i_t)$$

En appliquant cette observation aux modèles M_{impl} et M_{abs} on obtient :

$M_{impl} = I_1 || \ldots || I_n$ et
$M_{abs} = A_1 || \ldots || A_n$

Par application de la démarche de décomposition, on peut construire les composants :

$G_1 = I_1 || A_2 \ldots A_n$
$G_2 = A_1 || I_2 || A_3 \ldots A_n$
$\ldots$
$G_n = A_1 || \ldots A_{n-1} || I_n \ldots$

et vérifier que $\forall i = 1..n : G_i \preceq_r A_i$. On note toutefois qu'aucun de ces buts de preuve intermédiaires ne satisfait le critère de correspondance totale des variables d'état, mais que pour tout i, les variables d'état de A_i peuvent être retrouvées parmi les variables d'état de G_i. L'approche directe consisterait donc à calculer l'ensemble des états atteignables de G_i, puis à raisonner sur les relations de transition de G_i et de A_i, conformément à l'équation (18).

Cependant, en utilisant les règles de raisonnement résumées par le tableau 3.1, il est possible de pousser la décomposition encore plus loin, grâce au théorème suivant :

Théorème 3.1 *Soient $M_{impl} = I_1 || \ldots || I_n$ un modèle symbolique et $M_{abs} = A_1 || \ldots || A_n$ sa spécification abstraite, tels que M_{impl} est défini par les fonctions de transition $Fi_1, \ldots, Fi_n$ et M_{abs} est défini par les fonctions de transition $Fa_1, \ldots, Fa_n$. Soit $\chi_k, k = 1..n$ la fonction caractéristique de l'ensemble des états atteignables du composant G_k.*

Si :

$$\chi_k \rightarrow \exists \langle ND_r \rangle : Fi_k \Leftrightarrow Fa_k$$

alors $G_k \preceq_r A_k$.

Demonstration

La k-ième étape dans la preuve de raffinement par décomposition a pour but d'établir si :

$G_k \preceq_r A_k$

Comme le critère de correspondance totale entre les variables d'état de G_k et de A_k n'est pas satisfait, nous allons recourrir à un artifice compositionnel. Prenons un ensemble de composants $U_1, \ldots, U_{k-1}, U_{k+1}, \ldots U_n$ qui devrait nous permettre d'établir que :

1. $G_k \preceq_r U_1 || \ldots U_{k-1} || A_k || U_{k+1} \ldots || U_n$, où
   $G_k = A_1 || \ldots A_{k-1} || I_k || A_{k+1} \ldots || A_n$, puis
2. $U_1 || \ldots U_{k-1} || A_k || U_{k+1} \ldots || U_n \preceq_r A_k$.

La deuxième étape est trivialement vraie, quels que soient les composants U_i, grâce au lemme 1 du tableau 3.1. Il reste à établir, à partir de l'hypothèse, que la première étape est vraie. On obtiendrait ainsi par transitivité que $G_k \preceq_r A_k$.

Soient $ND_u^i, i = 1..n$ un ensemble de pseudo-entrées. Soient $U_i, i = 1, n$ les composants dont la relation de transition est $Tu_i = s_i' \Leftrightarrow ND_u^i$. Conformément à la définition de la relation de raffinement, $G_k \preceq_r (U_1 || \ldots U_{k-1} || A_k || U_{k+1} \ldots || U_n)$ si :

$$\chi_k \rightarrow \exists \langle ND_r \rangle, \exists ND_u^1, \ldots ND_u^n :$$
$$(Fa_1 \Leftrightarrow ND_u^1) \ldots . (Fi_k \Leftrightarrow Fa_k) \ldots . (Fa_n \Leftrightarrow ND_u^n)$$

Le terme $(Fi_k \Leftrightarrow Fa_k)$ ne dépend pas des variables ND_u^i. En distribuant les quantificateurs existentiels compte-tenu de cette observation, l'expression ci-dessus s'écrit :

$$\chi_k \rightarrow \exists \langle ND_r \rangle : \big((Fi_k \Leftrightarrow Fa_k) \; . \; \exists ND_u^1, \ldots ND_u^n :$$
$$(Fa_1 \Leftrightarrow ND_u^1) \ldots . (Fa_{k-1} \Leftrightarrow ND_u^{k-1}) . (Fa_{k+1} \Leftrightarrow ND_u^{k+1}) \ldots (Fa_n \Leftrightarrow ND_u^n)\big)$$

La deuxième partie de cette expression est trivialement vraie : chaque fonction de transition abstraite est comparée par rapport à une pseudo-entrée quantifiée existentiellement. Donc, l'expression ci-dessus se réduit à :

$$\chi_k \rightarrow \exists \langle ND_r \rangle : (Fi_k \Leftrightarrow Fa_k)$$

ce qui correspond à notre hypothèse de départ. On obtient ainsi que :

$G_k \preceq_r U_1 || \ldots U_{k-1} || A_k || U_{k+1} \ldots || U_n$

Donc,par transitivité (tenant compte des étapes 1 et 2) on obtient $G_k \preceq_r A_k$.
□

Ce théorème prouve que si on arrive à comparer une à une et successivement les fonctions de transition de l'implémentation et du modèle abstrait, alors on peut déduire le raffinement.

Cette technique est proche de celle implémentée dans la preuve de raffinement de l'outil SMV de Cadence.

Exemple : application à la preuve de raffinement $ARB' \preceq_r ARB_{abs}$. Nous reprenons ici la preuve de raffinement illustrée au paragraphe §2.4.3.2. Le modèle ARB' peut être décomposé en un produit de trois modèles séparés :

$ARB' = I_1 || I_2 || I_3$, où
I_1 est défini par la relation de transition $Ti_1 = (s_1' \Leftrightarrow F_t^1)$
I_2 est défini par la relation de transition $Ti_2 = (s_2' \Leftrightarrow F_t^2)$
I_3 est défini par la relation de transition $Ti_3 = (s_3' \Leftrightarrow F_t^3)$

De la même façon, le modèle abstrait peut être décomposé de la façon suivante :

$ARB_{abs} = A_1 || A_2 || A_3$, où

$Ta_1 = (s'_1 \Leftrightarrow Fa^1_t)$
$Ta_2 = (s'_2 \Leftrightarrow Fa^2_t)$
$Ta_3 = (s'_3 \Leftrightarrow Fa^3_t)$

Le but de preuve $ARB' \preceq_r ARB_{abs}$ se transforme donc en : $I_1||I_2||I_3 \preceq_r A_1||A_2||A3$.

Contrairement à l'exemple montré §2.4.3.2, nous allons décomposer la preuve de raffinement en sous-buts plus simples, en applicant le raisonnement "je suppose-tu garantis". Nous allons construire les modèles suivants :

$G_1 = I_1||A_2||A_3$,
$G_2 = A_1||I_2||A_3$ et
$G_3 = A_1||A_2||I_3$

En prouvant séparément que :

$G_1 \preceq_r A_1, \quad G_2 \preceq_r A_2$ et $\quad G_3 \preceq_r A_3$

nous pouvons conclure que $I_1||I_2||I_3 \preceq_r A_1||A_2||A3$. Donc $ARB' \preceq_r ARB_{abs}$. Ces preuves nécessitent le calcul des ensembles d'états atteignables pour G_1, G_2 et G_3. Compte-tenu du fait que ces composants combinent à la fois des parties du modèle abstrait et des parties de l'implémentation, leur représentation sera plus compacte et leur parcours symbolique plus aisé que celui de ARB'.

Compte-tenu du fait de la correspondence totale entre les variables d'état de ARB' et de ARB_{abs} il est possible de prouver le raffinement en raisonnant successivement sur les fonctions de transition. On a :

$\chi_1 \rightarrow \exists ND_1, ND_2 : Fi_1 \Leftrightarrow Fa_1$
$\chi_2 \rightarrow \exists ND_1, ND_2 : Fi_2 \Leftrightarrow Fa_2$
$\chi_3 \rightarrow \exists ND_1, ND_2 : Fi_3 \Leftrightarrow Fa_3$

où χ_1, χ_2, χ_3 représentent les ensembles d'états atteignables de G_1, G_2 et G_3. Ceci permet de conclure que $G_i \preceq_r A_i$ pour $i = 1..3$, et donc $ARB' \preceq_r ARB_{abs}$
□

Plusieurs outils de vérification universitaires [61, 7] ainsi qu'industriels [17] supportent la décomposition et le raisonnement "je suppose-tu garantis". Ils permettent également de confirmer si une décomposition est valide (présence ou non de cycles dans l'enchaînement du raisonnement). Par ailleurs, ce raisonement est également supporté par le standard EDA pour la spécification formelle SUGAR v2.0 [4].

3.2.2 Méthode ascendante

Lorsqu'un système complexe n'exhibe aucune propriété hierarchique, la seule approche de réduction appliquable systématiquement est la réduction par cône d'influence. Cependant, la pratique montre que cela peut ne pas être suffisant.

Une méthode empirique de réduction consiste à démarrer avec un système complexe mais non hiérarchique et à exploiter sa spécification ainsi que les constructions HDL de "haut niveau" utilisées pour sa description, afin de déduire un ensemble de *blocs fonctionnels*. On démarre ainsi avec une implémentation, et on tente de déduire ses principales fonctionnalités. Les différentes approches descendantes peuvent être appliquées au niveau des blocs fonctionnels ainsi obtenus. Bien que fastidieuse, cette méthode s'est révélée utile dans un contexte industriel [33]. Elle sera illustrée dans le prochain chapitre.

3.2.3 Exploiter la symétrie

La symétrie est une propriété d'ordre structurel d'un système. Une définition informelle de cette notion s'énoncerait de la façon suivante : deux éléments d'un système sont symétriques si et seulement si le fait de les permuter ne change en rien le comportement du système.

En pratique, la symétrie se décline de plusieurs manières. Soit $M = \langle X, O, S, T, \chi_0 \rangle$ un modèle symbolique. Dans ce contexte, nous allons considérer les variables X, O, et S. comme des ensembles ordonnés :

$X = \langle x_1, \ldots x_i, \ldots x_j, \ldots x_m \rangle$
$O = \langle o_1, \ldots o_h, \ldots o_k, \ldots o_p \rangle$
$S = \langle s_1, \ldots s_n \rangle$

Les cas suivants sont rencontrés fréquemment :

- symétrie entre deux variables d'entrée x_i et x_j de M. Soit $M' = \langle X', O, S, T, \chi_0 \rangle$, tel que $X' = \langle x_1, \ldots x_j, \ldots x_i, \ldots x_m \rangle$. Les variables d'entrée x_i et x_j sont symétriques si et seulement si $M \equiv M'$. Par exemple, les opérateurs booléens de base satisfont cette propriété de symétrie par rapport aux entrées ;
- symétrie structurelle entre les cônes d'influence de deux variables de sortie o_h et o_k. Construisons le modèle $M'' = \langle X'', O'', S'', T, \chi_0 \rangle$ tel que $O'' = \langle o_1, \ldots o_k, \ldots o_h, \ldots o_p \rangle$. Les cônes d'influence des variables o_h et o_k sont permutés. Cette permutation entraîne des conséquences sur l'ordre des variables dans X'' et S''. Les variables o_h et o_k sont symétriques si et seulement si $M \equiv M''$;
- de façon similaire, on peut évoquer la symétrie entre les cônes d'influence de deux variables d'état, s_i et s_j ;
- symétrie entre deux composants d'un modèle hiérarchique.

Lorsque deux composants d'un modèle sont symétriques, ils satisfont les mêmes propriétés au sein de ce modèle. Il suffit par conséquent de vérifier l'un des deux pour pouvoir déduire les propriétés de l'autre. Lorsque l'on évoque la symétrie structurelle entre les cônes d'influence de deux variables de sortie, il est possible de procéder à une réduction du système en se basant sur des sous-ensembles isomorphes. En d'autre termes, lorsqu'un ensemble est symétrique, n'importe lequel de ses éléments est représentatif vis-à-vis des autres. On peut donc réduire cet ensemble à un unique élément. Il est important de noter qu'en général, les propriétés de symétrie constituent une connaissance à priori du concepteur sur son système.

De nombreuses approches tentent de détecter la symétrie au sein d'un modèle à travers une étape d'analyse [20, 36] et d'en déduire le *modèle quotient* : une version réduite du modèle à vérifier, telle que chaque ensemble contenant des objets symétriques est reduit à un unique représentant.

Nous considérons toutefois que la connaissance à priori sur des propriétés de symétrie du modèle donne à ce jour les meilleurs résultats. Dans ce sens, une approche très intéressante a été proposée par Ip et Dill dans [24]. La méthode proposée s'appuie sur la notion d'*ensembles scalaires*[4]. Deux variables appartenant à un ensemble scalaire peuvent être permutées sans aucun changement sur le comportement du système. Une variable

[4] de l'anglais : *scalarset*

appartenant à un ensemble scalaire est nommée *variable scalaire*.

Lors de la conception, on affirme que certains sous-ensembles de variables (d'entrée, d'état ou de sortie) sont a priori symétriques : ce seront des ensembles scalaires. Cette approche de conception induit une contrainte sur le cône d'influence structurel du modèle résultant :

- une dépendance fonctionnelle entre deux variables scalaires peut exister uniquement si ces deux variables appartiennent à des sous-ensembles scalaires de même cardinal ;
- une variable non-scalaire ne peut pas dépendre d'une variable scalaire ;
- en revanche, une variable scalaire peut dépendre d'une variable non-scalaire.

A partir d'un modèle M et d'une partition de X, O et S en sous-ensembles scalaires et non-scalaires, tout en respectant les contraintes ci-dessus, il est possible de construire un modèle quotient : chaque ensemble scalaire est réduit à un seul élément représentatif. Ainsi, pour un ensemble scalaire de variables de sortie, on effectue d'abord cette réduction, puis on construit le cône d'influence pour la variable retenue.

Le raisonnement basé sur la symétrie sera illustré dans le prochain chapitre.

3.3 Approche comportementale

Nous avons montré qu'une spécification d'environnement constitue un élément essentiel pour la vérification d'un système par des techniques de décomposition structurelle. L'expérience montre que plus un composant à vérifier est complexe, plus la description de son environnement est complexe. Cette progression se poursuit à mesure que l'on se rapproche du niveau hiérarchique le plus "haut" correspondant à l'intégralité du système. C'est à ce niveau qu'une exception se produit. En règle générale, un système doit être *robuste* : l'environnement ne doit pas être en mesure de conduire le système à travers une séquence d'états vers une configuration où une erreur se produit. Cependant, la concision de la spécification d'environnement pour le système intégral est souvent contrecarée par la complexité du système lui même. Il est rare qu'à ce niveau, la vérification de modèles puisse aboutir. C'est pourquoi la spécification d'environnement est une étape fréquemment utlisée dans le cadre de la décomposition structurelle.

3.3.1 Retour sur la spécification d'environnement

Nous avons vu que les spécifications opérationnelles sont un moyen suffisamment flexible et efficace pour modéliser un environnement. Cependant, l'ajout d'une spécification d'environnement est souvent synonyme d'ajout de complexité dans le modèle symbolique pour la preuve.

Différentes méthodes de modélisation d'environnement ont été présentées dans la section §2.3.4, et classées par ordre de préférence par rapport à un critère d'expressivité. Si on tient compte de la complexité potentielle ajoutée au modèle symbolique pour la preuve, ce classement se trouve changé. Ainsi, ce sont les spécifications opérationnelles qui sont les plus pénalisantes.

En même temps, les spécifications de contraintes d'équité ont bien souvent une forme relativement simple et ne rajoutent pas, en elles mêmes, de complexité à la représentation

symbolique. Cependant, ce sont les algorithmes de preuve qui doivent en tenir compte qui sont très complexes, car ils effectuent un calcul de point fixe imbriqué.

Il est courant d'écrire une spécification d'environnement sous forme d'invariant : $G\ p$ où p est un prédicat booléen, exprimé sur les variables d'entrée-sortie du modèle à vérifier. Si p est une expression booléenne quelconque elle peut être utilisée comme hypothèse de départ, pour chaque calcul d'image ou de pré-image. Par exemple, un pas de calcul d'image tenant compte du fait que les valeurs des entrées satisfont le prédicat p s'écrit de la façon suivante :

$$\chi_{\mathrm{IMG}(\chi_E,p,T)}(\langle s'_n\rangle) = \exists\langle x_m\rangle \in \mathbb{B}^m, \exists\langle s_n\rangle \in \mathbb{B}^n : T(\langle x_m\rangle,\langle s_n\rangle,\langle s'_n\rangle).\chi_E(\langle s_n\rangle).p(\langle x_m\rangle) \quad (24)$$

Cette forme de spécification est préférable à l'écriture d'une spécification opérationnelle, car elle n'ajoute pas de variables supplémentaires au modèle symbolique. Dans beaucoup de cas, le "et" logique entre p et la relation de transition T engendre une simplification importante du modèle, mais toutefois non systématique.

Un cas bien plus intéressant se produit lorsque p se présente sous la forme d'un produit de variables d'entrée, complémentées ou non. Dans ce cas, les variables du sous-ensemble $Xc \subseteq X$ de cardinal $c \leq m$ sont remplacées par des constantes booléennes $\langle a_c\rangle \in \mathbb{B}^c$. La relation de transition du système prend la forme :

$$T(\langle x_{m-c}\rangle \bullet \langle a_c\rangle, \langle s_n\rangle, \langle s'_n\rangle) \quad (25)$$

qui revient à effectuer un cofacteur approprié de T par rapport aux variables de Xc. L'expression du cofacteur d'une fonction booléenne par rapport à une variable est systématiquement plus simple que l'expression de la fonction de départ.

3.3.2 La partition comportementale

Le mécanisme du cofacteur d'une fonction booléenne par rapport à une variable d'entrée est très utile dans certains cas, pour spécifier des *scénarios d'exécution* pour un système donné. La partition comportementale consiste à décomposer l'évaluation d'une spécification logique en autant de buts de vérification intermédiaires que de scénarios d'exécution identifiés. Chacun des sous-buts affirme que la propriété initiale est vraie sous l'hypothèse d'un scénario unique. Si tous les sous-buts sont prouvés, alors la propriété initiale peut être également vraie, quel que soit le scénario qui se produit.

Intuitivement, un scénario d'exécution désigne un ensemble de comportements du système. Cette notion a un caractère empirique, bien plus facile à illustrer grâce à l'exemple suivant.

Soit f une formule temporelle, affirmant que toutes les transactions sur un port de communication d'un système M sont correctes. Le but de preuve $M \models f$ peut être divisé en deux sous-buts distincts, selon les scénarios suivants : (1) seules les transactions de type "lecture" sont permises et (2) seules les transactions de type "écriture" sont permises. Si ces deux sous-buts sont vérifiés, on peut conclure que toutes les transactions sont correctes sur ce port. Dans chacun des deux sous-buts, la variable d'entrée indiquant la direction du transfert est contrainte à une valeur constante. Ceci permet d'obtenir un modèle sous-jacent plus compact, grâce à l'élimination d'une variable dans la relation de transition de M.

Cette méthode est applicable au port de communication de M, vu en tant qu'entité séparée. Si on prouve que les transferts en lecture et les transferts en écriture sont corrects, alors on peut conclure que les transferts sur ce port sont corrects. Dans le cas général toutefois, ce raisonnement n'est pas suffisant et doit être utilisé avec précaution. Prenons un exemple où le séquencement "transaction de lecture - transaction d'écriture" est significatif par rapport la validité d'une propriété f'. Le fait que f' soit vraie à la fois lors d'une transaction de lecture et d'écriture n'implique pas qu'elle soit vraie dans le cas d'un séquencement. Le fait de déterminer si ce genre de séquencement est significatif relève d'une connaissance à priori sur le modèle à vérifier.

3.4 Extension de la partition comportementale : partition fonctionnelle grâce à la simulation symbolique

Le fait d'établir un scénario d'exécution est un mécanisme très utile, mais qui se révèle insuffisant dans certains cas, à cause du caractère statique de cette technique. En effet, l'application d'un scénario d'exécution fige le système dans une certaine configuration propre au scénario spécifié. Il existe certains systèmes dont le comportement est caractérisé par la *reconfiguration dynamique* de leur *mode opératoire*. Dans le cas d'un tel système, à un instant donné, certains composants peuvent être qualifiés d'*actifs*, alors que d'autres sont *inactifs*. Pendant le fonctionnement, l'environnement peut ordonner l'activation ou la désactivation d'un composant. Cette opération nécessite l'application d'une séquence spécifique de valeurs d'entrée. Certaines propriétés du système n'ont de sens que *dans un certain mode opératoire, une fois que cette séquence a été exécutée*. A cause de son caractère dynamique, la spécification d'un mode opératoire dépasse les capacités de la partition comportementale.

Une approche compositionnelle permettant de profiter de ce schéma d'exécution particulier combine deux techniques : la *simulation symbolique* et la vérification de modèles [34]. La simulation symbolique permet d'amener le système dans un mode opératoire attendu, grâce à l'exécution d'une séquence partiellement symbolique pour les variables d'entrée. Ensuite, la vérification de modèles permet de vérifier la correction du système fonctionnant dans le mode opératoire atteint.

Dans la suite, la notion de mode opératoire sera présentée et illustrée. Ensuite, on définit formellement l'étape de simulation symbolique et on présente le raisonnement compositionnel basé sur la spécification de modes opératoires.

3.4.1 Modes opératoires

Nous nous intéressons à la vérification des systèmes qui peuvent être visualisés comme une interconnexion de composants hiérarchiques ou de blocs fonctionnels (si le système n'a pas de hiérarchie explicite). A un moment donné, un composant du système peut être :

- actif, lorsqu'il réagit au comportement de son environnement ;
- inactif lorsqu'il est dans son état initial, et qu'aucune transition n'est autorisée. En pratique, un composant inactif ignore son environnement ou bien répond à toute sollicitation avec un code d'erreur.

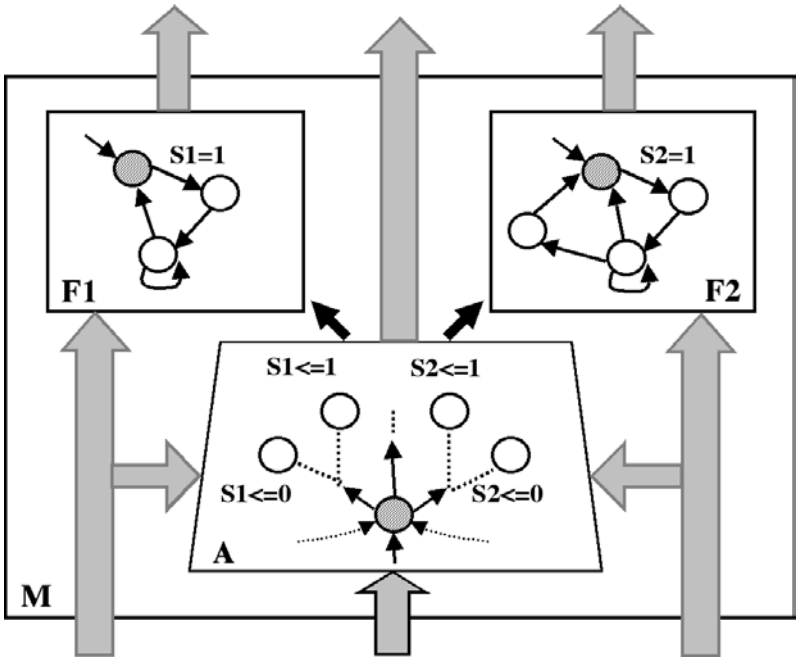

FIG. 3.3 – Modèle M contenant trois composants : F_1, F_2 et A

Une représentation schématique d'un tel système est donnée dans la figure 3.3. Le système M implémente trois composants : F_1, F_2 et A. Le composant A analyse le comportement de l'environnement et active F_1 ou bien F_2, ou l'un après l'autre, en affectant les variables internes S_1 et S_2. Un comportement similaire et dual permet de réinitialiser S_1 et S_2. Une fois activé (la transition S_1 devient active), le composant F_1 commence à réagir en conformité avec son environnement et peut éventuellement retourner dans son état initial. Tant que A ne réinitialise pas S_1, F_1 peut continuer son exécution. Ce même comportement s'applique à S_2 et F_2. Les composants F_1 et F_2 peuvent intéragir lorsqu'ils sont tous les deux actifs. Dans la suite, par souci de simplicité, nous considérons que seul F_1 est actif.

La vérification de M doit traiter séparément les composants F_1, F_2 et A. Les composants F_1 et F_2 ne peuvent être vérifiés à l'échelle de M. Le cône d'influence associé à F_1 ou à F_2 inclut forcément la description symbolique de A et serait de ce fait trop complexe. Il serait pourtant satisfaisant de pouvoir prouver qu'une fois F_1 activé, en supposant qu'il reste actif pour toujours, son comportement est correct. Une stratégie efficace de vérification appliquable aux composants tels que F_1 et F_2 doit permettre d'éviter l'inclusion de A dans le modèle symbolique pour la preuve.

Le comportement de M peut être *séquentiellement décomposé* et visualisé comme la concaténation de deux scénarios distincts :

- une séquence initiale de stimuli en entrée, qui *active* le composant F_1 ou F_2 ;
- un mode de fonctionnement "normal" : certains composants sont actifs, et réagissent par rapport à leur environnement.

La stratégie de preuve pour M suit cette particularité de fonctionnement. On décompose la vérification en deux étapes :

- activation du composant F_1 ; pour cela on exécute une séquence de *simulation symbolique* de M. Concrètement, cela revient à utiliser le document de spécification pour trouver une séquence de valeurs explicites qui, appliquée à certaines entrées de M, mène le composant A vers un état dans lequel S_1 est vrai. Cette étape spécifie un mode opératoire de M, dans lequel F_1 est actif et F_2 est inactif ;
- le mode de fonctionnement "normal" vient d'être atteint. Optionnellement, spécifier un scénario d'exécution permettant de figer M dans ce mode opératoire. Dans notre exemple, cela revient à contraindre les variables d'entrée de M de manière à empêcher le composant A de réagir par rapport à son environnement. Ainsi, l'environnement ne peut plus commander M pour altérer le mode opératoire courant. Dans le modèle symbolique résultant, F_1 restera actif pour toujours (et on peut tenter de le prouver). Ensuite, on vérifie l'ensemble des propriétés relatives au fonctionnement de F_1.

L'application de ces étapes permet de rendre le composant A superflu pour la vérification de F_1. La simulation symbolique démarre avec l'état initial de M ainsi qu'avec sa relation de transition et calcule un nouveau modèle, avec un nouvel état initial. La spécification ultérieure d'un scénario d'exécution permet d'obtenir et une nouvelle relation de transition réduite.

3.4.2 Définitions

Afin de donner un cadre formel à la notion de simulation symbolique que nous utilisons, nous allons partitionner l'ensemble X des variables d'entrée de M en deux sous-ensembles disjoints : $X = X_s \cup X_c$. L'ensemble X_s, de taille s représente le sous-ensemble des variables d'entrée qui restent symboliques. L'ensemble X_c, de taille c, contient les variables d'entrée de M auxquelles on affecte des valeurs booléennes explicites ($s+c=m$). Soient $\langle x_s \rangle$ et $\langle x_c \rangle$ les vecteurs obtenus par concaténation des éléments de X_s et X_c. A chaque élément $xc \in X_c$ on affecte une valeur booléenne conformément à une fonction d'association donnée :

$$Ass : X_c \to \mathbb{B} \tag{26}$$

Par analogie, on note $\langle Ass(X_c) \rangle$ le vecteur de constantes booléennes associé à $\langle x_c \rangle$.

Définition 3.1 (Pas de simulation symbolique) *Un pas de simulation symbolique SS est l'image d'un ensemble d'états E par rapport à la relation de transition T de M, dans le cas où toutes les variables d'entrée appartenant à X_c sont affectées avec des valeurs constantes, données par la fonction Ass. La fonction caractéristique χ_{SS} de cette image est donnée par l'expression :*

$$\chi_{SS(T,\chi_E,X_s,Ass)}(\langle s'_n \rangle) = \exists \langle x_s \rangle, \exists \langle s_n \rangle \in \mathbb{B}^n : T(\langle x_s \rangle \bullet \langle Ass(X_c) \rangle, \langle s_n \rangle, \langle s'_n \rangle) \ . \ \chi_E(\langle s_n \rangle) \tag{27}$$

On remarque que $SS(T, \chi_E, X_s, Ass) \subset \textsc{Img}(\chi_E, T)$. Il existe un cas particulier où

$$SS(T, \chi_E, X_s, Ass_0) = \textsc{Img}(\chi_E, T)$$

lorsque l'ensemble support de la fonction Ass_0 est vide ($X = X_s$).

Plusieurs pas de simulation symbolique peuvent être appliqués successivement, en utilisant à chaque fois une partition des entrées diffférente. Lors de chaque pas de simulation i, une partition d'entrée X_c^i spécifie quelles sont les variables d'entrée qui seront affectées à des valeurs constantes, données par $Ass^i(X_c^i)$. La simulation symbolique proprement dite s'effectue en appliquant un pas de simulation SS à k reprises, à partir de l'état initial de M :

$$\begin{aligned} Pas_1 &= \chi_{SS(T,\chi_0,X_s^1,Ass^1)} \\ Pas_{i+1} &= \chi_{SS(T,Pas_i,X_s^{i+1},Ass^{i+1})} \end{aligned}$$

3.4.3 Calcul du modèle réduit

C'est l'expression Pas_k qui caractérise l'ensemble des états initiaux du modèle réduit obtenu après k étapes de simulation symbolique. Ensuite, on applique une $k+1$-ième partition sur l'ensemble X. Le but de cette étape est de permettre la spécification d'un scénario d'exécution particulier, à appliquer une fois que la séquence de simulation symbolique a été exécutée. Bien qu'optionnelle, cette dernière étape est très importante pour l'obtention d'un modèle symbolique réduit. L'efficacité de cette réduction dépend à la fois du choix adéquat de X_c^{k+1}, ainsi que du choix de la fonction d'association Ass^{k+1} qui permet d'affecter des valeurs constantes à chaque élément $xc^{k+1} \in X_c^{k+1}$. La fonction Ass^{k+1} constitue une association *permanente*. Ceci permet de mettre en place un scénario d'exécution *après* les k pas de simulation symbolique et utilisable pendant l'étape de vérification proprement-dite.

L'algorithme permettant de calculer le modèle réduit est présenté dans l'Algorithme 3.1 ci-après.

En définissant une fonction d'association pour chaque pas de simulation, cet algorithme permet d'exécuter n'importe quelle séquence de simulation symbolique. En particulier, il permet de mettre en place un mode opératoire. Les expressions χ_r et T_r représentent l'état inital et la relation de transition réduite correspondant au mode opératoire atteint. Ces résultats constituent le modèle symbolique réduit, qui sera le point de départ pour l'évaluation d'une formule de logique temporelle.

Remarque. En pratique, il est nécessaire de restreindre la simulation symbolique de M au cône d'influence (structurel ou fonctionnel) de la propriété temporelle que l'on souhaite évaluer. Pour une formule temporelle f quelconque, il conviendrait d'appliquer tout d'abord l'algorithme "cone d'influence(FS_M, FT_M, f)" où FS_M est l'ensemble des fonctions de sortie de M et FT_M est l'ensemble des fonctions de transition de M. C'est seulement après que l'on applique l'algorithme "calcul modèle réduit" sur un modèle déjà réduit, obtenu à partir de l'ensemble $RESULTAT$, qui est le cône d'influence fonctionnel de M par rapport à f.

L'expérience montre qu'un grand nombre de modes opératoires peuvent être déduits directement du document de spécification (informel) du système à vérifier. Il arrive même parfois que la séquence de simulation nécessaire pour activer ou désactiver un mode opératoire soit fournie par la spécification. D'une manière plus générale, il est raisonnable

Procedure calcul modèle réduit (χ_0, T, k, $Ass^1 \dots Ass^{k+1}$)

nouveau : ensemble d'états atteints en un pas ;

courant : ensemble d'états déjà atteints ;

i $\leftarrow 1$;

courant $\leftarrow \chi_0$;

nouveau $\leftarrow \chi_0$;

while i $\leq$ k **do**

nouveau $\leftarrow$ SS(T, courant, X_s^i, Ass^i) ;

courant $\leftarrow$ nouveau $/s_i$ substitué à s'_i, pour chaque $s_i \in S$ et $s'_i \in S'$;

i $\leftarrow$ i + 1 ;

end while

$\chi_r \leftarrow$ courant ; `/* la relation de transition réduite,` T_r `est le cofacteur de` T `par rapport aux variables affectées par` Ass^{k+1} `: */`

$T_r \leftarrow T|_{\langle Ass^{k+1} \rangle}$

Algorithme 3.1: Algorithme de simulation symbolique

de croire que le concepteur connaît les séquences de simulation qui doivent être utilisées pour passer d'un mode opératoire à un autre. En contre-partie, en absence d'une telle connaissance, il est risqué de construire une telle séquence de façon empirique, sous peine de résultats de vérification incohérents.

La méthodologie de vérification basée sur la simulation symbolique est basée sur les étapes suivantes :

1. écrire une spécification logique f pour F_1 ;
2. écrire une séquence de simulation symbolique de longueur k, permettant d'activer F_1 ;
3. simuler symboliquement M sur k pas à partir de cette séquence ;
4. vérifier si le mode opératoire ciblé a bien été atteint. Dans notre exemple, si ce mode opératoire est identifié par $S_1 = 1$ alors on vérifie que dans l'état χ_r, atteint par simulation symbolique, on a : $M \models (S_1 = 1)$. Cela revient à évaluer l'expression $\chi_r \rightarrow (S_1 = 1)$;
5. spécifier éventuellement un scénario d'exécution final, à appliquer après la simulation symbolique ;
6. sur le modèle réduit ainsi obtenu, $M_r = \langle X, O, S, \chi_r, T_r \rangle$ vérifier que $M_r \models f$.

Bien qu'empirique, cette technique se montre particulièrement utile en pratique. Dans certains cas, elle peut s'appliquer de façon systématique. En effet, nombreux sont les systèmes qui utilisent une séquence d'initialisation. La plupart des propriétés intéressantes n'ont de sens qu'une fois le système initialisé. La simulation symbolique est une solution naturelle et efficace, permettant de prendre en charge la séquence d'initialisation et d'amener le

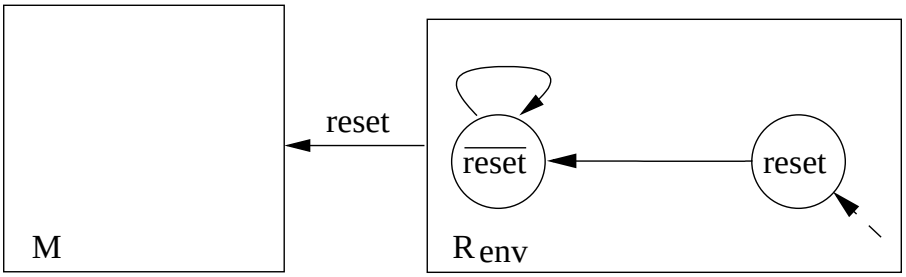

FIG. 3.4 – Définition d'une séquence d'initialisation pour M, grâce à une spécification opérationnelle R_{env}

système dans le mode de "fonctionnement normal". La plupart des outils industriels de vérification de modèles offrent la possibilité de spécifier une telle séquence d'initialisation. Cependant, cette séquence est représentée sous la forme d'une machine d'états (spécification opérationnelle d'environnement) qui sera composée avec le système à vérifier.

Comparée à cette méthode utilisée à grande échelle, la simulation symbolique apporte des avantages considérables. Le fait de composer le système avec une spécification opérationnelle a pour effet d'augmenter sa complexité, ce qui peut avoir des conséquences négatives sur l'efficacité de la preuve. En revanche, l'emploi de la simulation symbolique agit dans le sens opposé : non seulement elle ne complique pas le modèle symbolique, mais chaque séquence de simulation symbolique constitue une opportunité pour construire un modèle réduit.

Exemple. La grande majorité des circuits intégrés numériques possèdent une variable d'entrée dont le nom, *reset*, est presque consacré. Cette variable permet de remettre le système dans son état initial lorsqu'elle reçoit une séquence de valeurs adaptée. Par exemple, *reset* est actif pendant un certain temps, puis il devient inactif et le reste. Une telle séquence permet de placer le système dans un mode de fonctionnement dit "normal". La figure 3.4 illustre une spécification opérationnelle qui met en œuvre ce comportement. Le modèle symbolique resultant R_{env} est composé avec le système M et lui indique ainsi une séquence d'initialisation à respecter. Si f est une spécification logique de M, alors il est nécessaire de prouver que $M||R_{env} \models f$. Le modèle $M||R_{env}$ est nécessairement plus complexe que M.

D'un autre côté, cette séquence peut être exécutée grâce à la simulation symbolique. On spécifie :

$$\begin{array}{rcl} X_c & = & \{reset\} \\ Ass^1 : X_c \to \mathbb{B} & \text{telle que} & Ass^1(reset) = 1 \\ Ass^2 : X_c \to \mathbb{B} & \text{telle que} & Ass^2(reset) = 0 \\ Ass^3 : X_c \to \mathbb{B} & \text{telle que} & Ass^3(reset) = 0 \end{array}$$

La séquence de simulation symbolique est de longueur $k = 2$. Le dernier pas sert à établir un scénario d'exécution, une fois que l'initialisation a été effectuée, selon lequel *reset* reste

inactif pour toujours. Le modèle symbolique résultant aura une relation de transition de la forme :

$$T_r = T(0 \bullet \langle x_s \rangle, \langle s_n \rangle, \langle s'_n \rangle)$$

où $x_s \in X_s$. La représentation de T_r est forcément plus compacte que celle de T, grâce au cofacteur par rapport à la variable d'entrée *reset*. Ceci entraine des conséquences bénéfiques pour l'efficacité de l'évaluation de f à l'aide des algorithmes classiques de point fixe.
□

La partition fonctionnelle grâce à la simulation symbolique se montre particulièrement efficace lorsque le système vérifié n'est pas hiérarchique. Dans l'impossiblité de recourir à une decomposition structurelle, on peut effectuer une partition fonctionnelle, en spécifiant un ensemble de modes opératoires intéressants.

Cette stratégie est nouvelle et n'a pas encore fait l'objet d'une mise en œuvre au sein d'un outil industriel de vérification de modèles. Nous avons développé un prototype qui la met en œuvre et qui a été implanté au sein de l'outil NuSMV [18]. Son utilité sera illustrée dans le chapitre suivant, sur un exemple industriel.

Retour sur l'exploitation de la symétrie Aussi bien la partition comportementale que la partition fonctionnelle permettent d'annuler l'impact de certaines parties du système vérifié sur son comportement. Une conséquence importante de cette situation est qu'un ou plusieurs sous-ensembles du système vérifié deviennent symétriques. Cette propriété de symétrie est caractéristique du scénario ou du mode opératoire atteint, et ne peut pas être établie à l'échelle globale du système vérifié. C'est pourquoi nous appellerons cette propriété *symétrie fonctionnelle*.

Dans le contexte des techniques et des outils disponibles actuellement, la symétrie fonctionnelle n'est pas le résultat d'une analyse automatique du système vérifié, mais plûtôt une connaissance à priori du concepteur. Cette notion sera elle aussi illustrée dans le chapitre suivant.

3.5 Conclusion

L'efficacité des techniques de vérification symbolique de modèles diminue de façon exponentielle à mesure qu'augmente la taille du modèle symbolique analysé. C'est pourquoi les différentes approches de décomposition existantes constituent un outil extrêmement précieux pour la vérification d'un système. Parmi les techniques survolées dans ce chapitre, c'est la méthode "je suppose-tu garantis" qui est à la fois la plus puissante et la plus mature à l'heure actuelle. Ensuite, en fonction des possibilités, la partition comportementale et fonctionnelle sont elles aussi des techniques très utiles.

Quant à la preuve de raffinement, il ne s'agit pas encore d'une technique suffisamment mûre pour être utilisée de façon systématique dans un cadre industriel. Ceci est principalement dû au besoin d'une grande interaction avec l'utilisateur.

Ironie du sort, ou fait infortuné dans la vie de l'ingénieur de vérification, la décomposition structurelle vient seulement appuyer une démarche *positive* de vérification : "oui, la propriété spécifiée est bien vraie". La technique "je suppose-tu garantis" ne permet pas de

trouver des erreurs de conception, mais plûtôt de s'assurer de leur absence (dans la mesure des propriétés qui ont été écrites). Cependant, il est raisonnable de croire que :

- le nombre des erreurs de conception présentes dans la description d'un système reste largement supérieur au nombre des propriétés écrites et que l'on arrive à prouver ;
- à partir de ce postulat, la confiance dans la robustesse du système décrit s'acquiert à mesure qu'on arrive à détecter et corriger un certain nombre d'erreurs. Pour cela, il faut obtenir une réponse "négative", accompagnée d'un contre-exemple permettant de comprendre l'erreur.

Or, l'approche "je suppose-tu garantis" est très mal adaptée pour cette démarche. En effet, le fait qu'une formule temporelle soit fausse à l'échelle locale d'un composant peut être effectivement dû à une erreur de conception, mais aussi à une spécification d'environnement insuffisante. En règle générale, il est nécessaire de valider le contre-exemple obtenu à l'échelle globale du système vérifié, afin de savoir s'il est réaliste ou s'il est uniquement dû à un environnement trop peu contraignant. Cette validation fait nécessairement appel à des outils de simulation.

Chapitre 4

Mise en œuvre et Application

4.1 Introduction

La vérification formelle de propriétés temporelles est une technique relativement récente, qui est seulement en cours d'adoption au sein des cercles de développement industriels. La conception des systèmes discrets et en particulier des circuits éléctroniques numériques, continue à s'appuyer sur la notion de *description exécutable*. Les dernières décades ont vu la création de plusieurs langages, spécialement adaptés à la description de comportements *séquentiels*, *concurrents* ainsi que des *actions de synchronisation*, spécifiques aux systèmes numériques. Deux langages de description ont fait l'objet d'une standardisation par l'organisme IEEE et sont utilisés à grande échelle dans des contextes industriels : le langage VHDL [77] et le langage Verilog [78].

Pour un circuit décrit en VHDL ou en Verilog, la méthode de vérification la plus répandue et la mieux rodée reste la simulation. De nombreux outils de simulation très performants sont commercialisés et restent au cœur du développement des circuits.

Dans la suite de ce chapitre, nous nous consacrons exclusivement aux descriptions de circuits écrites en VHDL. Ce langage est supposé connu et le lecteur peut se référer à des ouvrages à caractère pédagogique pour de plus amples informations sur les langages de description de matériel [5, 82].

L'élaboration d'un circuit numérique est tributaire d'une démarche dont certaines étapes sont entièrement automatiques. On démarre à partir d'une description exécutable et on en extrait la description d'un réseau de portes électroniques. Ce résultat constitue un rapprochement important de la mise en œuvre physique du circuit. Cette opération porte le nom de *synthèse*. Plusieurs contraintes doivent être respectées pour que la synthèse puisse aboutir. Les constructions VHDL employées doivent pouvoir être mises en correspondance avec les éléments de base d'un circuit numérique physique. Actuellement, le niveau de description intitulé *transert de registres* (RTL[1]) est intensivement utilisé par les concepteurs et sert d'entrée aux outils de synthèse automatique.

Compte-tenu de ces réalités qui se dégagent du contexte de la conception des circuits industriels, il est apparu nécessaire de rendre possible la vérification formelle de propriétés

[1] en anglais : *Register Transfer Level*

à partir de la même description que celle qui sert d'entrée aux outils de simulation et de synthèse automatique.

Au niveau descriptif (VHDL) et pour des raisons liées à l'implémentation physique d'un circuit, les transitions d'états sont déclenchées par un événement marquant le passage du temps, provoqué périodiquement par un mécanisme d'horloge. Ainsi, lorsqu'une seule horloge est présente, on peut associer un changement d'état au sein du modèle FSM avec le chargement de tous les éléments mémorisants du circuit modélisé. Dans ce contexte, la sémantique de simulation VHDL peut être simplifiée, en considérant comme cycle de simulation un cycle d'horloge.

Les constructions VHDL permettant de décrire des circuits synthétisables ont été regroupées dans un sous-ensemble qui a été standardisé par l'organisme IEEE [1]. Le respect des règles d'écriture préconisées par ce standard garantit la conformité entre le modèle décrit par le concepteur et celui obtenu après l'opération de synthèse. Cette conformité se décline en termes de résultats de simulation identiques, mais aussi de façon plus stricte, en termes de modèles logiques équivalents.

La suite de ce chapitre fera un rappel des principaux styles de modélisation préconisés par le sous-VHDL standard pour la synthèse. On discutera ensuite de la sémantique de ces mécanismes de description pour la preuve formelle de propriétés. Une des réalisations de ce travail de thèse a été l'étude de la sémantique du sous-ensemble VHDL pour la synthèse RTL. On décrira ensuite une contribution à l'implémentation d'un outil effectuant l'extraction d'une FSM à partir d'une description VHDL synthétisable, en vue de l'utilisation de certains outils de vérification du domaine public. La stratégie de vérification basée sur la simulation symbolique a également fait objet d'une mise en œuvre au sein d'un outil de vérification.

4.2 Le standard IEEE 1076.6 pour la synthèse RTL

En plus de poser des restrictions sur les constructions VHDL à utiliser, le standard IEEE pour la synthèse identifie des styles d'écriture, reconnaissables syntaxiquement, et qui permettent de modéliser des comportements *combinatoires* ou *mémorisants* synchronisés par l'horloge, en accord avec la sémantique de simulation du langage VHDL. La notion de comportement combinatoire est synonyme de traitement instantané. C'est le cas de toute fonction booléenne, quelle que soit sa complexité. On considère que tout changement de valeur sur une ou plusieurs de ses entrées est immédiatement repercuté sur la valeur de la fonction. Quant au comportement mémorisant, il a pour unique but de modéliser les changements d'état du modèle à travers le chargement des mémoires à chaque occurrence d'un front d'horloge.

Pour décrire ces comportements, le standard IEEE définit deux classes principales de processus VHDL :

- les processus VHDL combinatoires contiennent du code VHDL procédural, satisfaisant obligatoirement plusieurs conditions. Tout d'abord, la fin de l'exécution de ce code doit être statiquement prévisible. Ensuite, tous les signaux VHDL lus par un processus combinatoire, doivent apparaître dans la liste de sensibilité ; ceci garantit l'exécution du processus à chaque événement (changement de valeur) qui se produit

sur un de ces signaux lus. Ensuite, s'il existe une exécution d'un processus combinatoire qui affecte explicitement une valeur à un signal VHDL, alors toutes les exécutions possibles de ce processus doivent affecter le même signal au moins une fois. De la même manière, toute variable interne à un processus combinatoire doit être affectée avant d'être lue. Les effets de mémorisation entre deux exécutions ne sont pas admis ;

- les processus VHDL mémorisants contiennent également du code procédural. L'exécution de ce code est rythmée par l'horloge du système. Lors de chaque nouvel événement qui se produit sur le signal d'horloge, le processus est exécuté une fois. Seul le signal d'horloge contrôle l'exécution d'un processus mémorisant. Une restriction s'applique également dans ce contexte de description. Ainsi, la fin de l'exécution d'un processus mémorisant doit être elle aussi statiquement prévisible. Les objets (variables ou signaux) affectés au sein d'un processus mémorisant peuvent engendrer des éléments mémorisants.

  Une description de nature combinatoire est tolérée au sein d'un processus VHDL mémorisant et ceci dans l'unique but d'exprimer un mécanisme asynchrone d'initialisation. En pratique, ce mécanisme est très important pour la définition de valeurs initiales des éléments mémorisants. Le standard IEEE pour la synthèse RTL préconise d'ignorer toute spécification d'une valeur initiale pour les signaux d'une description, sauf s'ils figurent au sein d'un paquetage VHDL.

Les éléments mémorisants définis par un comportement mémorisant synchronisé sur horloge sont connus sous le nom de *bascules*. Toutes les bascules chargent leur nouvelle valeur lorsqu'un front (montant ou descendant) se produit sur le signal d'horloge. Durant le fonctionnement du circuit, toutes les bascules changent de valeur simultanément, lorsque cet événement se produit.

Exemple. La figure 4.1 montre une description VHDL simulable et synthétisable, pouvant engendrer le modèle $CANAL_i$, composant de l'arbitre modulaire introduit au Chapitre 2 de ce document. Cette description fait appel à la fois aux notions de comportements mémorisants et combinatoires.

Les processus VHDL étiquetés *fonction_de_transition* et *fonctions_de_sortie* sont combinatoires. Ils mettent en œuvre le calcul de la fonction de transition et des fonctions de sortie du modèle $CANAL_i$. Les transitions du modèle sont décrites par le processus mémorisant *transition*. A chaque front montant du signal d'horloge, la valeur courante de Fm_t est affectée au signal sm. Cette affectation prend effet en temps négligeable par rapport à la durée d'un cycle d'horloge. Un éventuel changement de valeur de sm entraîne l'exécution des processus combinatoires *fonction_de_transition* et *fonctions_de_sortie* qui calculeront des nouvelles valeurs pour les signaux Fm_t, om, $prio\_s$ et adm_s.

On remarque que la valeur initiale du signal sm, correspondant à l'état initial du composant $CANAL_i$, est spécifiée lors de la déclaration de ce signal. Cette construction n'est pas conforme au standard IEEE. La synthèse ignore cette valeur initiale et génére pour sm une bascule dont la valeur intiale est la valeur par défaut du type *bit*, qui est 0. Dans le cas de cet exemple, les deux valeurs coïncident.

□

```
entity canal is
  port (
    x, prio_e, adm_e  : in  bit;        -- ports d'entrée
    om, prio_s, adm_s : out bit;        -- ports de sortie
    clk               : in  bit);       -- signal d'horloge
end canal;
architecture archi of canal is
  signal sm : bit := '0';               -- variable d'état de CANAL
  signal Fm_t : bit;                    -- fonction de transition
begin  -- archi
     fonction_de_transition: process (x, prio_e, adm_e, sm)
     begin
       Fm_t <= x and prio_e and (adm_e or sm);
     end process fonction_de_transition;

     fonctions_de_sortie: process (Fm_t, prio_e, x, sm)
     begin  -- process calcul_fonctions_de_sortie
       om     <= Fm_t;
       prio_s <= prio_e or (x and not sm);
       adm_s  <= x and prio_e;
     end process fonctions_de_sortie;

     transition: process
     begin
       wait until clk = '1';
       sm <= Fm_t;
     end process transition;
end archi;
```

FIG. 4.1 – Modélisation VHDL de niveau RTL synthétisable pour le composant $CANAL_i$ du modèle ARB_{mod}

L'impact sur la vérification formelle du traitement standard des valeurs initiales se traduit par un état initial par défaut, éventuellement différent de celui que le concepteur a souhaité exprimer, ce qui peut influencer la cohérence des résultats de vérification. Pour palier cet inconvénient, certains outils de vérification acceptent de s'écarter du standard afin de traiter les spécifications de valeurs initiales.

Le standard IEEE préconise un style d'écriture des comportements mémorisants, mettant en œuvre un mécanisme dynamique de spécification des valeurs initiales pour les éléments mémorisants. La remise à zéro se fait de manière *asynchrone*, c'est à dire indépendamment du signal d'horloge, qui est censé synchroniser l'ensemble des éléments mémorisants du circuit. Un port d'entrée VHDL (ou plusieurs si nécessaire) est dédié à l'initialisation de ces éléments.

```
transition_async_reset: process (clk, rst)
begin  -- process async_reset
  if rst = '0' then                    -- condition de remise à zéro
                                       -- (active à l'état bas)
    sm <= '0';
  elsif clk'event and clk = '1' then   -- front montant de l'horloge
    sm <= Fm_t;
  end if;
end process transition_async_reset;
```

FIG. 4.2 – Variante du process *transition* avec remise à zéro asynchrone

Exemple. La description d'un comportement mémorisant avec remise à zéro est schématisée par le processus VHDL de la figure 4.2. Le processus VHDL *transition_ async_ reset* est la variante avec remise à zéro asynchrone du processus *transition*. L'instruction conditionnelle *if* qui figure dans ce processus est obligatoire. Ainsi, lorsque le signal de remise à zéro des mémoires *rst* est actif[2] le signal *sm* est initialisé avec la valeur $'0'$. Ce signal est prioritaire par rapport à l'horloge : aussi longtemps qu'il reste actif, l'horloge est ignorée. La désactivation du signal *rst* permet au processus *transition_ async_ reset* d'exécuter l'ensemble des instructions synchronisées sur chaque front montant d'horloge. En règle générale, le signal *rst* a un comportement très particulier : il est actif au début, pendant un temps très court, correspondant à la "mise sous tension" du circuit, puis il devient inactif et le reste, permettant ainsi au circuit de fonctionner normalement, sans autre remise à zéro.
□

Le caractère asynchrone du signal de remise à zéro n'est pas conforme à l'hypothèse temporelle en vigueur sur le signal d'horloge. Conformément à cette hypothèse, les changements de valeur de tous les signaux VHDL, ports d'entrée compris, ne sont pris en compte qu'une fois par cycle d'horloge. Ce problème peut être traité à plusieurs niveaux.

Tout d'abord, on peut exprimer une restriction sur le signal de remise à zéro : on suppose que ses changements de valeur se conforment aux contraintes temporelles imposées par l'horloge. Ainsi, son comportement asynchrone est simplifié et resynchronisé sur l'horloge.

La mise en œuvre de cette hypothèse se fait à travers la réécriture du processus *transition_ async_ reset* en déplaçant la condition de remise à zéro des bascules à l'intérieur de la condition de synchronisation sur l'horloge. Le processus VHDL *transition_ sync_ reset* représenté dans la figure 4.3 illustre ce comportement. L'application de cette restriction nécessite seulement une manipulation syntaxique très simple. Elle peut donc facilement être rendue systématique, soit lors du codage soit en tant qu'étape préliminaire à l'extraction d'un modèle FSM pour la preuve.

Une deuxième manière de traiter la présence d'un signal de remise à zéro asynchrone s'appuie sur le concept d'extraction avec *horloge explicite*, qui sera illustré dans la prochaine section.

[2] les signaux de remise à zéro sont généralement considérés actifs lorsqu'ils ont la valeur '0'

```
transition_async_reset: process (clk)
begin  -- process async_reset
  if clk'event and clk = '1' then    -- front montant de l'horloge
    if rst = '0' then                -- condition de remise à zéro
                                     -- resynchronisée
      sm <= '0';
    else
      sm <= Fm_t;
  end if;
end process transition_async_reset;
```

FIG. 4.3 – Variante du process *transition* avec remise à zéro resynchronisée sur le signal d'horloge

4.3 Sémantique du sous-ensemble IEEE pour la preuve

Une fois le circuit modélisé en VHDL au niveau RTL, se pose le problème de l'extraction d'une FSM symbolique à partir de sa description. Cette FSM doit servir pour la preuve par vérification de modèles.

Le problème de l'extraction d'un modèle FSM à partir d'une description VHDL n'est pas nouveau. Les premiers travaux ayant pour but d'appliquer la vérification de modèles à des descriptions VHDL datent des années '93. Des travaux portant sur la sémantique d'un sous-ensemble VHDL pour la preuve ont été menés par Bull [29] et ont servi à la mise en œuvre de l'outil VFORMAL. D'une manière simultanée, Déharbe propose un sous-ensemble VHDL adapté pour la preuve par vérification de modèles [31, 30]. Il définit également la procédure d'extraction d'un modèle FSM à partir d'une description VHDL synchronisée par horloge.

Parallèlement, Encrenaz utilise le modèle des réseaux de Pétri pour la modélisation et la vérification formelle de descriptions VHDL [37]. Nous avons également étudié l'adéquation de ce formalisme pour la vérification de systèmes matériels, à travers une démarche de validation et de vérification de systèmes totalement asynchrones [14]. Dans notre contexte cependant, les réseaux de Petri semblent plus adaptés pour la formalisation de systèmes logiciels, et il a été difficile de créer un lien efficace entre la modélisation VHDL, le modèle des réseaux de Petri et celui des machines d'états finis, qui constituait notre représentation cible en vue d'utilisation d'outils industriels de vérification. Tous ces travaux ont posé les fondements de la mise en évidence d'un sous-ensemble VHDL adapté pour la preuve formelle.

Par la suite, les efforts se sont concentrés sur le traitement d'un sous-ensemble synthétisable de VHDL, proposé par SYNOPSYS et qui fait encore office de standard. Des concepteurs d'outils industriels de vérification, tels que Cadence et IBM, se sont appuyés sur ce sous-ensemble.

En parallèle, nos travaux se sont intéressés à l'extraction de modèles FSM à partir de

descriptions VHDL standard, en vue de l'utilisation de logiciels de vérification universitaires. Dans un premier temps, ce travail de thèse s'est consacré à la génération de modèles FSM à partir de descriptions VHDL conformes au standard IEEE pour la synthèse. Le format de représentation cible de la génération était BLIF-MV [57], langage d'entrée de l'outil VIS [45], développé à Berkeley. Ensuite, pour des raisons d'intérêt expérimental, nous avons adapté l'outil de génération VHDL - BLIF-MV à la génération des formats SMV, supportant à la fois la version Cadence et la version CMU de ce langage. L'intérêt de cette démarche est de rendre possible l'application des techniques expérimentales mises en œuvre dans l'outil SMV de Cadence ; par ailleurs, nous avons opté pour la gestion de la syntaxe CMU dans le but de pouvoir utiliser l'outil NuSMV [18] au sein duquel nous avons implémenté la synergie entre la simulation symbolique et la vérification de modèles.

4.3.1 Extraction d'un modèle symbolique

Cette opération a pour but de transformer une description VHDL en un modèle FSM booléen reconnaissable par les outils de vérification de modèles. Elle comporte plusieurs étapes importantes. Tout d'abord il est nécessaire d'analyser le code VHDL, afin de s'assurer qu'il est conforme au standard IEEE pour la synthèse RTL. Ensuite, l'extraction proprement-dite peut commencer. Elle est composée des étapes suivantes :

4.3.1.1 Détection du schéma de synchronisation

La détection du signal d'horloge du circuit se fait à travers l'analyse du code VHDL. Le standard préconise plusieurs styles d'écriture permettant de spécifier une synchronisation sur une horloge. Par exemple, pour exprimer une synchronisation sur front montant, on peut utiliser une parmi les écritures suivantes :

1. `wait until RISING_EDGE(clk);`
2. `wait until clk = '1';`
3. `wait until clk = '1' and clk'event;`
4. `wait until clk = '1' and not clk'stable;`

Hormis la deuxième variante, l'ensemble des écritures présentées ci-dessus peuvent être adaptées pour l'expression de la synchronisation à l'aide d'une instruction séquentielle if. La synchronisation sur front descendant s'exprime de façon similaire. A partir des schémas identifiés ci-dessus, il faut déterminer s'il y a un port d'entrée de la description qui est utilisé suivant une des façons indiquées ci-dessus. L'analyse doit également déterminer si l'ensemble de la description est synchronisé sur un front unique, ou bien si la synchronisation est mixte. Ensuite, il faut déterminer si le signal d'horloge est unique ou non. En effet, la présence de deux ou plusieurs ports d'entrée différents qui se revèlent chacun être un signal d'horloge a un impact très important sur l'extraction du modèle. Si ces horloges sont indépendantes, alors l'hypothèse d'horloge unique initiale est fausse et l'algorithme de simulation VHDL ne peut pas être simplifié au niveau cycle d'horloge. Les transitions du modèle extrait ne correspondront pas aux cycles d'horloge, mais au δ-cycle VHDL.

En pratique, cependant, on peut accepter des signaux d'horloge différents sous l'hypothèse que toutes les horloges existantes soient synchronisées sur une unique horloge de

base, appelée *horloge universelle*. Leur comportement est donc défini en termes de l'horloge universelle.

4.3.1.2 Détection des variables d'état

Ce problème consiste à trouver un ensemble d'objets VHDL (variables et signaux) modifiés au sein d'un processus VHDL, dont la valeur est nécessairement mémorisante. Cette déctection est appliquée aux processus combinatoires, afin de déterminer si leur exécution induit un effet de mémorisation non souhaité. Elle est également appliquée aux processus mémorisants, dans le but de construire une partie du modèle booléen cible : l'ensemble des variables d'état. Pour des raisons d'efficacité de la preuve formelle, cet ensemble doit être aussi réduit que possible. Tout objet (variable ou signal) affecté sous condition d'horloge n'est pas nécessairement une variable d'état. En règle générale, pour qu'une variable VHDL soit mémorisante, il faut que sa valeur soit lue en début d'un cyle d'horloge, avant d'être affectée. L'étape de détection dénombre tous les chemins d'exécution possibles entre deux instructions *wait*. En même temps, elle comptabilise les affectations et les références de chaque variable le long de chacun de ces chemins. Si une variable v est référencée le long d'un chemin d'exécution mais sans que cette référence soit précédée d'une affectation, alors v engendre une variable d'état dans le modèle booléen extrait. Dans le cas contraire, v engendre une variable interne à comportement combinatoire.

En ce qui concerne les signaux VHDL, affectés au sein de processus mémorisants, ils engendrent de façon quasi-systématique des objets mémorisants. Certaines optimisations simples sont possibles en cas de partage d'expressions avec d'autres objets mémorisants déjà détectés. Les détails de la réconnaissance des objets mémorisants ont fait l'objet de travaux parus dans [35, 32].

Les objets aggrégés VHDL nécessitent un traitement particulier dans le cadre de la détection des variables d'état. Lorsqu'ils sont traduits en équations booléennes, ils sont éclatés en éléments simples de type booléen. Cependant, en simulation, une propriété d'atomicité est rattachée aux tableaux et aux structures VHDL. Par exemple, lorsqu'un événement (changement de valeur) se produit sur un élément d'un objet aggrégé, on considère que l'ensemble de l'objet a changé. Il est donc nécessaire de déterminer, tenant compte de cette particularité, si la propriété de mémorisation doit obéir ou non à ce principe d'atomicité. En effet, au sein d'un processus mémorisant, il est tout à fait possible de lire un sous-ensemble strict d'un objet aggrégé avant que celui-ci soit affecté. Ainsi, seule une partie de l'aggrégé manipulé serait réellement mémorisante. Nous avons cependant choisi d'appliquer le principe d'atomicité à la propriété de mémorisation. Un objet aggrégé ne peut donc être partiellement mémorisant. Ce choix peut pénaliser l'efficacité de la preuve, car le modèle booléen obtenu peut contenir des variables d'état "parasites". Nous comptons cependant sur une démarche cohérente au niveau de la modélisation et recommandons au concepteur d'éclater les objets aggrégés en parties ayant des propriétés de mémorisation uniformes.

4.3.1.3 Extraction des fonctions de transition et de sortie

Cette opération doit s'appliquer sur un niveau de description contenant à la fois des comportements concurrents et mémorisants, décrits de manière procédurale. Dans un

contexte concurrent, les signaux VHDL ne peuvent être affectés qu'une seule fois. Si cette règle n'est pas respectée, il est nécessaire de faire appel au mécanisme des fonctions de résolution. Les fonctions de résolution définies par l'utilisateur ne font pas partie du sous-ensemble défini par le standard IEEE pour la synthèse. En conséquence, la génération des affectations concurrentes (en particulier des fonctions de sortie affectées dans un contexte concurrent) se fait à la volée, lors du parcours du code VHDL.

En revanche, l'affectation de signaux et/ou de variables dans un contexte procédural n'obéit pas à la règle d'affectation unique. Le concepteur peut écrire toute séquence d'instructions pourvu qu'il soit possible de prédire statiquement la fin de son exécution. Pour générer les fonctions de transition et de sortie dans ce contexte, il est donc nécessaire de comptabliser l'ensemble des affectations dont les signaux et les variables ont fait objet. Pour cela, on utilise le mécanisme des variables à affectation unique : chaque affectation d'une variable ou d'un signal engendre une étiquette numérotée. Lorsqu'une variable est référencée au sein d'une expression, elle est remplacée par la dernière étiquette générée. Dans le cas des signaux, la valeur du pilote est rendue effective seulement avec un retard d'un δ-cycle, conformément à la sémantique de simulation VHDL. Ainsi, lorsqu'un signal est référencé au sein d'une expression, on garde toujours sa valeur courante, et non pas la dernière valeur qui lui a été affectée. Les détails de cette génération sont également présentés dans [35].

Cette technique a été mise en œuvre au sein de l'outil `v2mv`, qui analyse une description VHDL, et qui la traduit en BLIF-MV, le format d'entrée de l'outil VIS.

La génération des fonctions de transition est étroitement liée au schéma de synchronisation retenu pour le circuit. Comme il a été rappelé précédemment, lorsque la description possède une horloge unique, et que tous ses éléments sont synchronisés sur le même front, il est possible de raisonner sur un algorithme de simulation simplifié, dont les cycles de simulation se superposent aux cycles d'horloge. C'est ce qui permet de s'abstraire du signal d'horloge lors de la génération du modèle FSM.

En revanche, le signal d'horloge doit rester explicite dès qu'un parmi les scénarios suivants se produit :

- le signal d'horloge apparaît explicitement dans la description ailleurs que dans les instructions de synchronisation définies par le standard ;
- la description est synchronisée de façon mixte. Certaines parties se synchronisent sur le front montant de l'horloge et d'autres parties sur le front descendant.

Lorsque l'horloge est implicite, on fait abstraction de la condition de synchronisation présente dans les processus VHDL mémorisants et on génére un modèle booléen avec des fonctions de transition sous la forme d'identités booléennes. Ainsi, si v est un signal VHDL, alors l'affectation VHDL sous condition d'horloge `v <= expression ;` engendre une variable d'état correspondant au signal v, ainsi qu'une identité booléenne de la forme $v' \leftrightarrow expr$ qui exprime le fait que la valeur de v coïncide avec la valeur de $expr$ à l'état précédent. Ceci se traduit en termes physiques par la présence d'une "bascule" (figure 4.4). En sortie de la bascule on représente la valeur courante de la variable d'état. A chaque front d'horloge donc, v est mis à jour avec la valeur de $expr$.

Lorsque le signal d'horloge doit être explicite, la FSM résultante doit contenir un mécanisme permettant de détecter l'occurrence d'un front d'horloge. La figure 4.5 illustre le

VHDL :

```
process (clk)
begin
 if clk'event and clk = '1' then
   v <= expr;
 end if;
end process;
```

expression Booléenne extraite et correspondance physique :

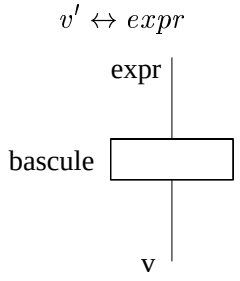

FIG. 4.4 – Extraction du modèle Booléen pour un signal VHDL affecté sous condition d'horloge : hypothèse d'horloge implicite

Modèle Booléen détecteur de fronts montants :

variable d'entrée : $\{clk\}$

variable d'état : $\{délai\}$

variable de sortie : $\{front\}$

fonction de transition :
$délai' = f_t(délai, clk) = clk$

fonction de sortie :
$front = f_o(délai, clk) = \overline{délai}\ .\ clk$

état initial : $délai =' 0'$

représentation physique :

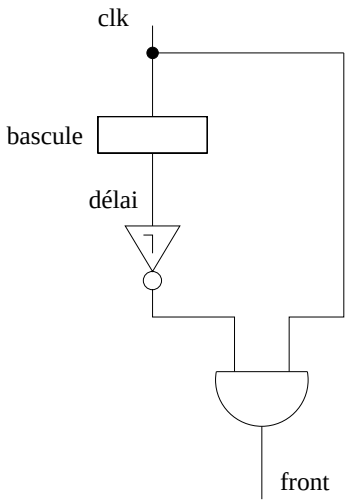

FIG. 4.5 – Modèle Booléen permettant de détecter l'occurrence d'un front montant sur le signal d'horloge

modèle booléen permettant détecter l'occurrence d'un front montant. Si la valeur courante du signal d'horloge est 1 et que sa valeur précédente, indiquée par la variable d'état *délai* est 0 alors un front montant vient de se produire.

Ainsi, lorsque l'horloge doit rester explicite, la même affectation VHDL engendre un modèle booléen plus complexe, faisant appel au mécanisme de détection du front montant

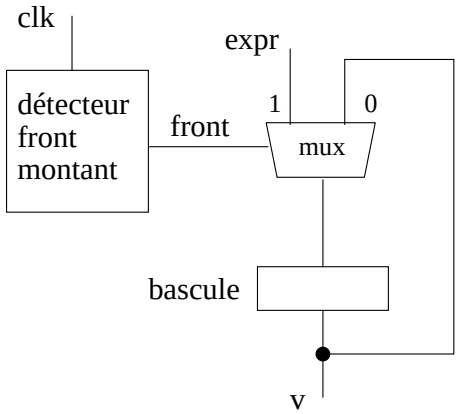

FIG. 4.6 – Extraction d'un modèle booléen avec horloge explicite à partir de l'affectation VHDL de la figure 4.4

de l'horloge. La figure 4.6 représente ce nouveau modèle booléen. Si un front montant vient de se produire, alors la valeur qui sera mémorisée dans v est celle de $expr$. Dans le cas contraire, v garde sa valeur précédente.

Ce schéma d'extraction est mis en œuvre au sein de certains outils de vérification de modèles acceptant des descriptions Verilog [45] ou VHDL [17].

Remarque. Lorsque le signal d'horloge du circuit analysé reste explicite, il est souvent nécessaire de le modéliser en tant que spécification opérationnelle d'environnement. Cette opération permet de spécifier la périodicité avec laquelle ce signal peut changer de valeur. Toute horloge explicite devient une variable d'entrée dont le comportement se décline en fonction des états du modèle FSM sous-jacent au circuit traité, qui constituent les pas de base de calcul de ce modèle. La durée de chaque phase de l'horloge doit donc couvrir une ou plusieurs transitions d'état, à l'échelle de ce modèle. Ce pas de calcul, qui équivaut à une transition dans la FSM, est appellé *"cranck"* ou *"tick"* dans le contexte de certains outils industriels de vérification de modèles.

4.3.2 Retour sur la sémantique du cycle de simulation

La présence du signal d'horloge au sein du modèle Booléen extrait à partir d'une description VHDL fait encore objet d'un débat, quant à la modélisation de ce signal et des objets mémorisants qu'il influence. Les choix qui ont été faits et mis en œuvre dans les outils entraînent à ce jour des divergences sémantiques entre le modèle booléen extrait et le modèle élaboré pour la simulation. Par exemple, le modèle booléen représenté dans la figure 4.6 introduit une erreur par rapport à la sémantique stricte de simulation du langage VHDL. Conformément à cette sémantique, l'affectation d'un signal dans un contexte mémorisant se fait lorsque le front montant vient de se produire. Le pilote de v est affecté avec la valeur courante de $expr$. Un δ-cycle plus tard, cette affectation doit se repercuter sur la valeur courante de v. Ce n'est pas le cas dans le modèle booléen qui a été extrait. Conformément à ce modèle, la valeur courante de v est mise à jour de manière décalée non

pas d'un δ-cycle mais d'un *"tick"*. Ainsi, un délai invisible en VHDL se retrouve traduit en un délai visible dans le modèle booléen extrait. Cet inconvénient a été considéré mineur, et les concepteurs l'acceptent sans rencontrer de problème dans la plupart des cas.

Nous avons cependant rencontré une situation dans une description VHDL industrielle, montrant les limites de cette approche.

Exemple. Considérons la description VHDL suivante :

```
.......
P1 : process(clk, rst)
     begin
       if rst = 1 then
         transfer <= IDLE;
       elsif clk = '1' and clk'event then
          if read_or_write = 1 then
             transfer <= READ;
          else
             transfer <= WRITE;
          end_if;
       end if;
     end process;
P2 : process(clk,transfer)
     begin
       if clk = '1' then
         transfer_type <= transfer;
       else
         transfer_type <= IDLE ;
       end if;
     end process;
......
```

Le processus VHDL $P1$ affecte sous front montant d'horloge le signal $transfer$, selon la valeur du signal $read\_or\_write$, dont le contexte d'affectation n'est pas explicité. A chaque front montant, le pilote du signal $transfer$ est affecté ; sa valeur courante est mise à jour au bout d'un δ-cycle. Quant au processus $P2$, il lit la valeur du signal $transfer$ et, lorsque l'horloge vaut $'1'$, il l'affecte de manière combinatoire au signal sortant $transfer\_type$. La simulation de cette description permet de constater qu'à partir du premier front montant de l'horloge, $transfer\_type$ contient toujours la dernière valeur affectée à $transfer$. Cette situation est illustrée par la partie haute de la figure 4.7.

En pratiquant l'extraction du modèle booléen selon la méthode suggérée par la figure 4.6, le δ-cycle qui sépare l'affectation de $transfert$ de sa mise à jour effective est rendu visible et égal à un pas de calcul FSM (*"cranck"*).

Dans notre exemple, le signal d'horloge a été modélisé en termes de ces pas de calcul de FSM : aussi bien sa phase haute que sa phase basse durent chacune pendant un pas. C'est ainsi que l'affectation séquentielle

```
transfer_type <= transfer;
```

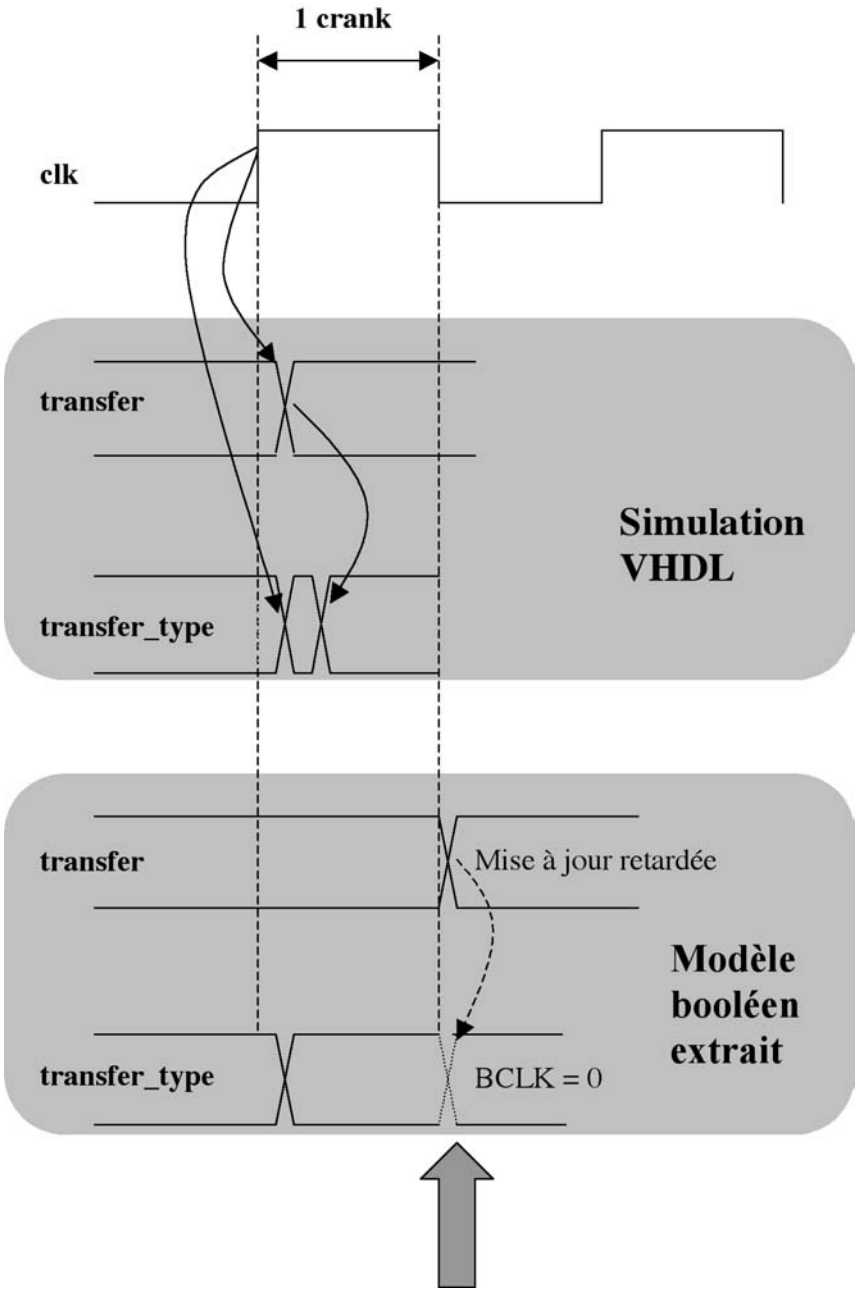

FIG. 4.7 – Divergences possibles entre la simulation VHDL et le modèle booléen extrait avec horloge explicite

ne pourra jamais lire la valeur la plus récemment affectée à $transfer$, contrairement à la réalité de la simulation, comme le montre la partie basse de la figure 4.7. De ce fait, ce modèle booléen n'est pas correct.

□

L'incohérence illustrée par l'exemple ci-dessus a pu être contournée par une modélisation spéciale du signal d'horloge. Dans un premier temps, on a diminué sa période en augmentant le nombre de pas FSM par phase. Ceci nous a permis de référencer et de distinguer, grâce à des variables d'état auxiliaires, le "début de la phase haute" pendant lequel le signal $transfert$ n'a pas encore été mis à jour, de la "fin de la phase haute" lors de laquelle la mise à jour et la propagation des valeurs attendues ont été effectuées. Cette solution n'est cependant ni élégante ni efficace, car elle introduit des variables d'état suplémentaires dans le modèle.

Une façon plus élégante de contourner ce problème consiste à s'écarter légèrement du modèle de simulation VHDL, en considérant que les valeurs échantillonnées par le front d'horloge sont celles qui étaient disponibles juste avant que ce front ait lieu. Au sein du modèle booléen extrait, on lit les valeurs des variables d'état et des entrées qui étaient disponibles avant l'occurrence du front d'horloge. Au même moment où le front se produit, les nouvelles valeurs des variables d'état deviennent effectives. Ce comportement se rapproche du schéma de mise à jour des valeurs effectives des signaux après l'occurrence du front d'horloge. Cette mise à jour est effectuée au bout d'un nombre fini de δ-cycles, étant donc observable en même temps que le front.

Cette approche possède toutefois le désavantage d'une modélisation booléenne non conforme à la sémantique de simulation du langage VHDL.

Le problème commun à ces deux approches de modélisation des affectation séquentielles est l'absence d'un mécanisme permettant d'accéder à deux informations distinctes : la valeur courante (mémorisée) du signal VHDL affecté, et sa nouvelle valeur, déclenchée par le front d'horloge, et qui est stockée dans le pilote du signal affecté. Cette nouvelle valeur se propage à travers toutes les dépendances combinatoires de la description avec un délai nul.

Il est utile de modéliser chaque affectation de signal sous front d'horloge par *deux* variables. Nous allons considérer dans la suite que v est un signal VHDL, affecté dans un contexte mémorisant. Ce signal sera modélisé par la variable v_p, permettant d'accéder à son pilote, et la variable v_{mem}, permettant d'accéder à sa valeur courante. La figure 4.8 illustre la modélisation booléenne du signal VHDL v. L'utilisation de ces deux variables obéit aux règles suivantes :

- lorsque le signal VHDL v est lu dans un contexte mémorisant, il sera remplacé dans le modèle booléen généré par la variable "valeur courante" v_{mem} ;
- lorsque le signal VHDL v est lu dans un contexte combinatoire, il sera remplacé dans le modèle booléen généré par la variable "pilote" v_p.

Ces règles sont nécessaires pour éviter l'apparition d'éventuels cycles combinatoires dans le modèle booléen extrait. Leur application permet de créer des points d'*observation* v_p au sein du modèle booléen extrait. L'utilisation de ces points d'observation permet de baser le raisonnement sur la sémantique de simulation VHDL lors de l'écriture d'une spécification temporelle.

Cette approche n'est pas essentiellement différente de celle présentée figure 4.6. La seule différence vient de la présence des points d'observation, utilisables dans des contextes d'affectation combinatoires. Cependant, son utilisation est limitée aux descriptions au sein desquelles les seules affectations combinatoires ont pour cible des ports de sortie. En effet,

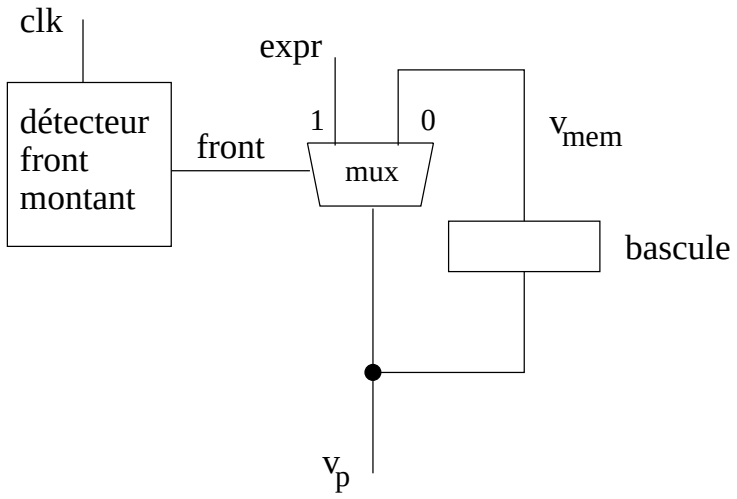

FIG. 4.8 – Modèle booléen avec horloge explicite offrant l'accès différencié (δ-cycle/cycle d'horloge) aux valeurs disponibles lors du chargement d'une bascule

compte-tenu de l'artifice mis en œuvre, aucune valeur combinatoire ne peut être réinjectée au sein d'un contexte mémorisant sous peine de perte de la cohérence du modèle. Il est pourtant possible, à travers une étape de réécriture qui se résume à une série de remplacements, de transformer toute description synthétisable de manière à la rendre conforme à ce critère. Ces remplacements ont pour but de s'assurer que chaque signal mémorisant ne dépend finalement que de signaux d'entrée et/ou d'autres signaux mémorisants. Ce type de manipulation peut avoir des conséquences néfastes sur la taille du code VHDL réécrit, car il peut engendrer la duplication de certaines expressions. En revanche, il n'aura pas de conséquence négative sur l'efficacité de la traversée symbolique du modèle FSM extrait.

4.3.2.1 Gestion des verrous

Certaines descriptions VHDL utilisent des éléments mémorisants particuliers connus sous le nom de *verrous*[3]. Les verrous sont des mémoires à caractère asynchrone : la mémorisation d'une nouvelle valeur peut avoir lieu indépendamment des fronts de l'horloge. En VHDL, un tel comportement conforme au standard pour la synthèse se décrit de la manière indiquée figure 4.9. Dès que le signal VHDL *chargement* prend la valeur $'1'$, le signal s est affecté avec la valeur courante de *donnee*. Tant que *chargement* $='1'$, tout changement de valeur qui se produit sur *donnee* se répercute avec un délai égal à un δ-cycle sur le signal s. On dit que le verrou s est transparent. Lorsque la valeur de *chargement* passe de $'1'$ à $'0'$, s se trouve *vérouillé* sur la dernière valeur présente sur *donnee* au moment où cet évènement se produit.

A cause du caractère asynchrone de cette construction, l'extraction d'un modèle booléen synchronisé par horloge à partir d'un code VHDL comportant des verrous est problématique. Il est possible, sous certaines conditions, de resynchroniser le comportement du

[3]plus couramment utilisés sous leur nom anglais : *transparent latch*

```
verrou: process (chargement, donnee)
begin
  if chargement = '1' then    -- condition de mémorisation
    s <= donnee;
  end if;
end process verrou;
```

FIG. 4.9 – Modélisation VHDL de niveau RTL synthétisable pour un verrou

```
verrou_rs: process (chargement, donnee)
begin
  if chargement = '1' then    -- condition de mémorisation
    s <= donnee;
  else
    s <= mem;
  end if;
end process verrou_rs;

memoire: process (clk)
begin
  if clk'event and clk = '1' then
    mem <= s;
  end if;
end process memoire;
```

FIG. 4.10 – Modélisation VHDL du verrou resynchronisé

verrou sur le signal d'horloge, en le modélisant à l'aide d'une bascule, tout en conservant l'équivalence par rapport au modèle de départ. La figure 4.10 illustre cette transformation. Le processus VHDL *verrou_rs* affecte la valeur du verrou s. En cas de transparence, il retransmet *donnee*. Sinon, s prend la dernière valeur mémorisée. La mémorisation est mise en œuvre à l'aide du signal VHDL auxiliaire *mem*.

Cependant, à cause du caractère synchrone conféré par cette transformation, le modèle obtenu ne pourra plus verrouiller les éventuelles valeurs transitoires disponibles uniquement entre deux fronts d'horloge. Ceci conduit très facilement à des résultats de simulation faux. Lors de l'étape de stabilisation qui a lieu entre deux fronts d'horloge, le signal *chargement* peut subir plusieurs mises à jour, engendrant la mémorisation d'une valeur transitoire dans le modèle de départ. La même mémorisation ne se produit dans le modèle resynchronisé qu'en présence d'un front d'horloge. Ces incohérences ne sont pas détectables statiquement [29] et rendent de ce fait l'utilisation des verrous très délicate.

Afin d'éviter la détection de ce genre d'incohérence au niveau δ-cycle VHDL, qui peut se révéler très coûteuse, il est plus intéressant d'utiliser des restrictions temporelles sur le comportement du signal *chargement*.

modèle Booléen :

variable d'entrée : $\{clk\}$

variable d'état : $\{mem\}$

variable de sortie : $\{s\}$

fonction de transition :
$mem' = f_t(mem, clk, donnee) =$
$clk.donnee + \overline{clk}.mem$

fonction de sortie :
$s = f_o(mem, clk, donnee) = f_t$

état initial : $mem =' 0'$

représentation physique :

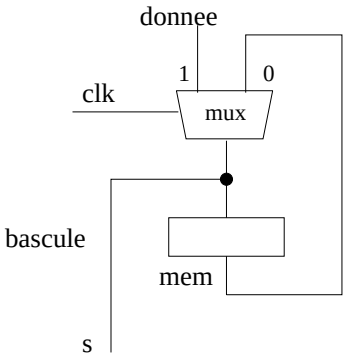

FIG. 4.11 – Modèle Booléen d'un verrou chargé par le niveau haut de l'horloge

Une restriction possible consiste à imposer que ce signal se stabilise au plus tard au bout du δ-cycle suivant le front d'horloge. Cette règle est suffisante pour s'assurer qu'aucune valeur transitoire n'est mémorisée. Le verrou résultant doit avoir des périodes de transparence et de verrouillage qui s'étendent sur au moins un cycle d'horloge.

En pratique, il est possible de recourir à une alternative de modélisation moins restrictive. Il s'agit d'utiliser le signal d'horloge même comme signal de chargement. Le verrou résultant serait transparent pendant une phase de l'horloge et verrouillé pendant la phase opposée. Cette solution impose de garder le signal d'horloge explicite lors de l'extraction d'un modèle booléen. Dans ce cadre, le comportement de l'horloge est défini au sein de l'environnement, en termes de pas de calcul de FSM (*ticks*). Compte-tenu de la définition de ce signal bien particulier, aussi bien la période de transparence que le verrouillage s'étendent sur des multiples de ce pas de calcul. L'utilisation des pas de calcul FSM permet également de spécifier une périodicité de mémorisation plus élevée que celle spécifiée par l'horloge.

Ceci permet d'écarter les incohérences dûes aux éventuels comportements transitoires du signal de chargement par rapport à la périodicité de mémorisation. Le modèle extrait de la description VHDL figure 4.9 est celui présenté dans la figure 4.11.

A notre connaissance, l'outil industriel de vérification de modèles RuleBase [59] supporte actuellement ce mécanisme. Les constructions mémorisantes sensibles aux niveaux sont également acceptées par l'outil FormalCheck, mais leur traitement ainsi que leur interprétation est limité.

Malgré l'existence de quelques règles permettant de modéliser des comportements sensibles aux niveaux, il est souhaitable d'éviter l'introduction de verrous au sein de la description. En effet, l'emploi de ces composants introduit des risques aussi bien lors de l'extraction d'un modèle booléen pour la vérification formelle, mais aussi lors de la synthèse classique. Les éventuelles erreurs liées à l'emploi des verrous sont principalement liées à des

problèmes de course critique entre signaux. Ce problème nécessite une analyse du circuit tenant compte des contraintes temporelles de propagation des signaux. A l'heure actuelle, ce genre d'erreurs sont très difficilement détectables.

Dans ce contexte, le fait d'employer des verrous entraîne une forte probabilité de divergence entre la sémantique du modèle booléen extrait pour la preuve et celle du réseau de portes synthétisé.

4.4 Mise en œuvre de la stratégie de vérification basée sur la simulation symbolique

Cette stratégie de vérification a été présentée dans le chapitre précédent. Elle a fait l'objet d'une mise en œuvre dans le but de la valider sur des exemples de taille industrielle. Les descriptions VHDL comportementales conformes au standard IEEE pour la synthèse RTL sont traduites dans la version CMU du langage SMV. Cette nouvelle représentation n'est autre qu'un modèle booléen de la description initiale.

L'utilisateur doit fournir la description du mode opératoire sous la forme d'une séquence de simulation symbolique, ainsi que l'ensemble des formules temporelles CTL qui doivent être satisfaites une fois le mode opératoire atteint. Accessoirement il peut être nécessaire d'inclure une spécification d'environnement. Celle-ci peut être écrite soit en VHDL, soit directement dans le langage SMV. A partir de ces données, l'outil NuSMV effectue la vérification, éventuellement après interaction avec le moteur de simulation symbolique que nous y avons récemment intégré (figure 4.12).

4.4.1 Description des séquences de simulation symbolique

Chaque pas de simulation symbolique commence par affecter des valeurs constantes sur un sous-ensemble des variables d'entrée du modèle vérifié. Afin de réaliser cette affectation, nous avons opté pour la mise en œuvre la plus directe, qui utilise le concept d'instanciation d'un composant. Cette opération est appliquée autant de fois que de pas de simulation nécessaires.

Exemple. Nous illustrons ci-dessous la description d'une séquence de simulation symbolique, telle qu'elle est fournie à l'outil NuSMV. Soit *arbitre(x1,x2,x3,reset,requete,donnee)* un composant et ses ports formels. Une séquence de simulation symbolique appliquée à cet arbitre sur 3 pas peut porter sur la remise à zéro des variables d'état. Ainsi, on affecte à la variable *reset* les valeurs $0, 1, 1$.

```
pas_1 : arbitre(x1,x2,x3,0,requete,donnee);
pas_2 : arbitre(x1,x2,x3,1,requete,donnee);
pas_3 : arbitre(x1,x2,x3,1,requete,donnee);
```

Au bout du troisième pas, on considère l'arbitre initialisé et on peut vérifier les propriétés relatives à ce mode opératoire bien particulier.
□

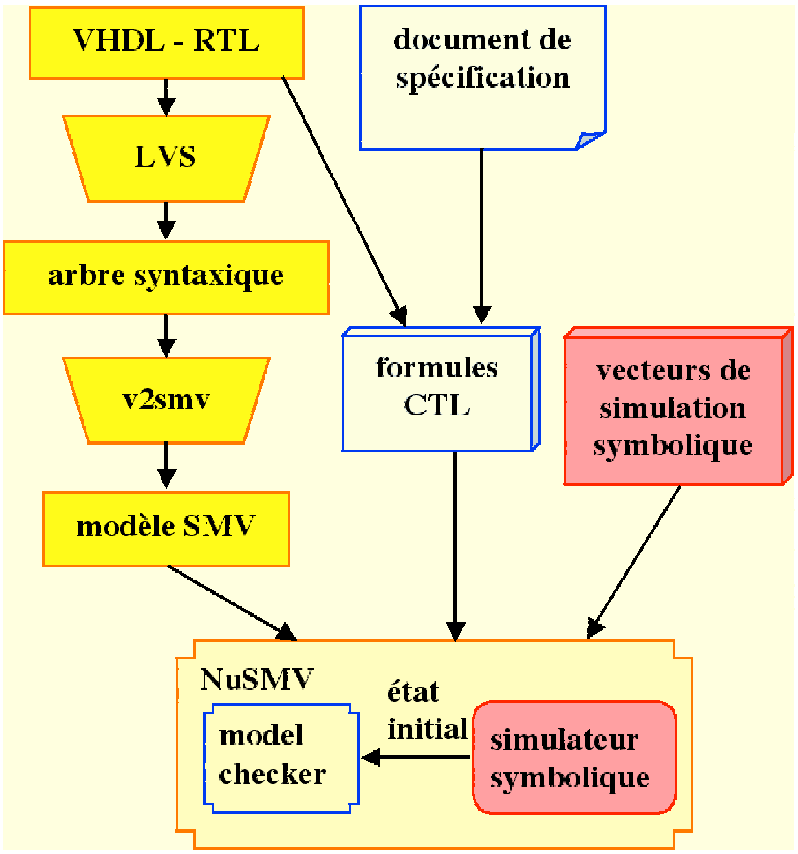

FIG. 4.12 – Mise en œuvre de la stratégie de vérification basée sur la simulation symbolique

Dans le contexte de la simulation symbolique, l'instanciation du composant simulé effectue l'affectation des variables d'entrée, puis calcule le modèle symbolique correspondant à cette affectation. L'appel de la procédure *Simulation symbolique* se fait au début de la vérification. La procédure lit la séquence de simulation qu'elle doit exécuter et en déduit sa longueur $k+1$.

Le modèle que l'on souhaite vérifier doit généralement être accompagné d'une spécification d'environnement. Cette spécification se fait de manière opérationnelle. On lit donc le fichier SMV *global*, contenant la spécification du modèle à vérifier ainsi que la spécification de son environnement. Avant la construction d'un modèle booléen, on effectue une restriction du modèle global par rapport au cône d'influence induit par la propriété f à vérifier.

L'exécution de la simulation symbolique alterne une étape de construction d'un modèle symbolique avec un calcul d'image. Hormis dans le cas où $i = 0$, le résultat obtenu lors du pas i, sert de point de départ lors du pas $i+1$. Une fois cette exécution terminée, on évalue la propriété f par vérification symbolique classique. Ces étapes sont explicitées par l'algorithme 4.1.

Compte-tenu de la présence de valeurs constantes à la place de certaines variables d'entrée, l'expérience montre que les calculs successifs des modèles symboliques préconisés

Procedure Simulation symbolique

```
/* Lecture des vecteurs de simulation symbolique */
/*  et construction de la liste liste_pas_sim */
/* Calcul de k, le nombre de pas de simulation à effectuer */
/* Déterminer mod_sim, le module sur lequel */
/*  s'applique la simulation symbolique */
/* Lecture du fichier SMV dans lequel mod_sim et son environnement */
/*  sont instanciés */
/* Déterminer f, la formule CTL à évaluer sur mod_sim */
/*  après la simulation symbolique */
/* Mise à plat de la hiérarchie */
/* Construction du modèle booléen restreint par rapport au  */
/* cône d'influence induit par f */
for i = 0..k-1 do
  /*  Exécution de la simulation symbolique. Les différents BDDs calculés
  sont pointés par les éléments du tableau pas(0..k-1) */
  /* Instanciation du i^ème pas :  */
  /*    mise à jour du modèle booléen restreint, tenant compte des valeurs
  constantes/symboliques spécifiées par liste_pas_sim(i) */
  if i == 0 then
    pas(i) = Image(état initial de mod_sim)
  else
    pas(i) = Image(pas(i-1))
    /* Effacer le BDD pointé par pas(i-1) */
  end if
end for
/* pas(k-1) pointe sur l'état initial du modèle */
/* après la simulation symbolique */
```

Algorithme 4.1: Algorithme générique de calcul d'un modèle réduit à travers une séquence de simulation symbolique

par cet algorithme sont beaucoup moins coûteux en ressources que le calcul du modèle symbolique global.

4.5 Application : vérification formelle d'un circuit industriel

4.5.1 Introduction

Tout au long de ce document, nous nous sommes penchés sur les différentes techniques et stratégies d'application de la vérification de modèles. Le but final de l'ensemble des manipulations présentées est de limiter l'utilisation des ressources afin de maîtriser la complexité de la vérification. Ce problème peut se produire sur des descriptions de taille réduite, qui engendrent parfois des modèles symboliques très volumineux, comme c'est le cas des circuits arithmétiques. Par ailleurs, dans un contexte industriel, on est souvent confronté à la vérification de descriptions très complexes, pour lesquels la construction même d'une relation de transition est impossible. Dans de telles situations, la technique de la vérification de modèles perd son caractère automatique. Seule une intéraction avec l'ingénieur de vérification peut l'aider à aboutir. Ces aspects ont été développés dans [33, 34].

Dans cette section, nous présentons un travail mené dans un contexte industriel. Le but de ce travail était d'appliquer certaines des techniques et stratégies présentées dans ce mémoire à la vérification de circuits réels, impossibles à vérifier par application directe des logiciels de "*model checking*".

Le travail de l'ingénieur de vérification doit tenir compte des contraintes suivantes :
- le langage de description imposé pour la description des circuits est VHDL ;
- aucune description n'est modifiable. Les éventuelles suggestions de modification doivent être adressées à l'équipe de conception.

Dans la suite de ce chapitre, nous présentons la vérification d'un circuit réel, composant de taille importante d'un "système sur une puce" : un contrôleur de cache d'instructions. Nous commençons donc par présenter l'architecture de ce module, ainsi que les caractéristiques du modèle VHDL utilisé pour sa description. Ensuite, on se penche sur la formalisation de la spécification de ce circuit. Seront ensuite présentées les différentes stratégies de preuve envisagées, ainsi que leurs résultats chiffrés, qui montrent l'importance de leur application sur l'efficacité de la preuve.

4.5.2 Etude de cas : un contrôleur de cache d'instructions

4.5.2.1 Présentation de l'architecture

Ce système connecte un processeur de signal (DSP) et un ensemble de bancs de mémoire interne (situés sur la même puce) ainsi que des bancs de mémoire externe (Figure 4.13). A chaque cycle d'horloge, le DSP peut envoyer une demande au contrôleur, par le biais du *bloc de chargement DSP*.

Le contexte de fonctionnement du contrôleur est sauvegardé au sein du *registre d'état interne*, qui a pour rôle d'enregistrer les données suivantes : quels sont les bancs valides ou invalides, où en est un transfert DMA[4], quel est le statut (actif ou inactif) du bloc de chargement DSP, ainsi que l'adresse de base de la page stockée dans chaque banc interne. Le contenu de ce registre est mis à jour par le DSP, à travers le port de commande.

[4]"Direct Memory Access", Mécanisme d'accès direct et concurrent à la mémoire externe.

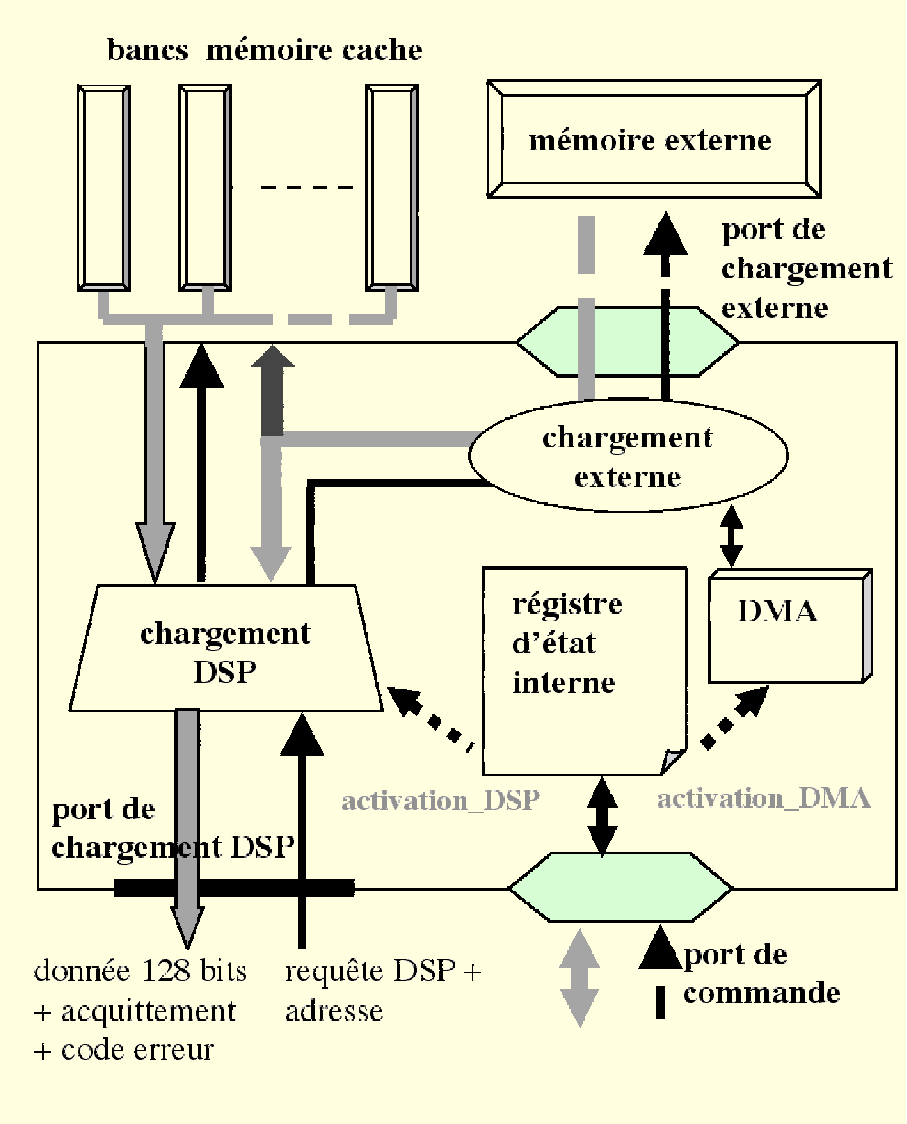

FIG. 4.13 – Architecture du contrôleur de cache

Lorsque le mot mémoire demandé par le DSP est présent dans un des bancs internes, il est renvoyé vers le DSP au bout de 3 cycles d'horloge (délai constant). Dans le cas où le mot demandé est absent de la mémoire interne, la requête est redirigée vers le *bloc de chargement externe*. Une requête de lecture est envoyée à travers le *port de chargement externe*. La réponse à cette requête peut arriver sous un délai arbitraire, en général très long.

Lorsque le bloc *chargement DSP* est inactif, il répond aux requêtes DSP avec un code d'erreur.

Contrairement aux mécanismes plus classiques, il n'y a pas de chargement automatique d'une nouvelle page mémoire contenant le mot demandé, lorsque celui-ci ne se trouve pas

dans le cache. Le rafraîchissement d'un banc doit être commandé explicitement par le DSP : à travers le *port de commande*, il doit envoyer une commande de téléchargement d'une nouvelle page mémoire. En parallèle, le *registre d'état interne* est mis à jour pour indiquer que le banc en cours de chargement n'est plus valide. Les demandes correspondant à un banc invalide reçoivent en réponse un code d'erreur. Le bloc DMA demande au bloc de chargement externe autant de transferts de mots que nécessaire pour remplir un banc. Lorsque le téléchargement est terminé, le banc est validé.

Pendant qu'un banc est chargé par le DMA, les autres doivent rester accessibles pour d'éventuelles requêtes de la part du DSP. En outre, si une requête correspond à une adresse externe, elle doit être retransmise vers le port de chargement externe. Ainsi, un transfert DMA et un chargement externe doivent partager le même port de manière équitable.

Les ports de commande et de chargement externe doivent obéir à un protocole de communication prédéfini. Dans ce contexte de développement, ce protocole définit une interface et des échanges "standard" à utiliser de manière systématique au sein de la description.

Plusieurs modes opératoires se dégagent de l'architecture de ce contrôleur de cache. Par exemple, le moteur DMA peut être actif, c'est à dire en train d'effectuer le chargement d'une page mémoire, pendant que la partie "chargement DSP" est inactive. D'autres combinations "actif/inactif" relatives à ces deux blocs sont également possibles. Tous ces modes opératoires partagent le même mécanisme d'activation : le DSP effectue une mise à jour de l'entrée appropriée au sein du registre d'état interne.

4.5.2.2 Le modèle VHDL

Le contrôleur de cache est implémenté en VHDL, au niveau transfert de registres (RTL). Dans ce circuit, aucune construction hiérarchique n'est utilisée. Les différents blocs (chargement DSP, DMA, etc.) sont implémentés à l'aide de processus VHDL combinatoires et séquentiels. Ainsi, il est difficile de raisonner indépendamment sur les différentes parties de ce système, par application de la décomposition hiérarchique.
Dans ce circuit, la partie contrôle est prédominante. Le bloc DMA implémente un calcul arithmétique : un compteur indique l'indice du dernier mot téléchargé. La borne supérieure du compteur reflète la taille d'un banc interne, qui est égale à 1024 mots. L'analyse d'atteignabilité de ce compteur est extrêmement inefficace, car elle nécessitera 1024 itérations, qui s'ajoutent à l'exploration du système global. Les objets vectoriels ne sont pas symétriques. Leurs différentes tranches suivent des chemins différents dans le circuit.

Les chiffres les plus représentatifs concernant ce système sont les suivants : 1500 lignes de code VHDL ; l'élaboration engendre un ensemble de 1000 registres ; les bus d'adresse ont 30 bits, et les bus de données ont 32 bits. La largeur du port côté DSP est de 128 bits.

4.5.2.3 Spécification formelle du contrôleur de cache

La spécification d'un circuit définit de manière implicite et informelle des propriétés que celui-ci doit satisfaire. Dans le cas idéal, l'ingénieur de vérification n'a pas besoin de connaître les détails de l'implémentation du circuit. Il lui suffirait de formaliser et vérifier l'ensemble des assertions contenues dans ce document. Si toutes ces assertions sont prouvées, le circuit peut être considéré comme étant "sans erreur" ; on dit que l'on a atteint

une couverture fonctionnelle raisonnable. Cependant, un document de spécification est rarement clair et exhaustif. Un plan de vérification se révèle souvent nécessaire, permettant de répertorier, sous forme d'assertions, tous les comportements décrits par la spécification, ainsi que de lever les éventuelles ambiguïtés introduites par une spécification informelle. La correction d'un circuit est examinée par rapport au plan de vérification. Ces assertions sont écrites soit en logique temporelle, soit sous la forme de spécifications opérationnelles, et constituent la spécification formelle du circuit.

Dans le cadre de ce document, nous ne présentons qu'une version partielle de la spécification formelle de ce contrôleur. Les assertions présentées concernent les fonctionnalités suivantes (figure 4.13) : le chargement DSP, le port de commande, le moteur DMA et le registre d'état interne.

Le chargement DSP Voici quelques aspects intéressants pour la vérification de cette fonctionnalité :

- (DSP-1) *délai de réponse constant* : lorsque le bloc "chargement DSP" est actif, chaque requête DSP est acquitée avec un délai constant de 3 cycles, si le mot demandé se trouve bien dans la mémoire cache.
- (DSP-2) *délai de réponse fini* : lorsque le bloc "chargement DSP" est actif, chaque requête DSP est acquitée dans un délai fini, si le mot demandé se trouve dans la mémoire externe.
- (DSP-3) *refus de service* : lorsque le bloc "chargement DSP" est inactif, chaque requête DSP est refusée avec un code d'erreur ;
- (DSP-4) *précédence de requête* : lorsque le contrôleur renvoie vers le DSP un mot extrait de la mémoire cache, alors il y a forcément eu une requête appropriée, trois cycles plus tôt ;
- (DSP-5) *correction du mot transmis* : tout mot d'instruction lu dans la mémoire cache arrive inchangé au DSP.

Les propriétés DSP-1,2,3,5 se décrivent de manière très naturelle à l'aide de la logique temporelle linéaire ou arborescente. Il s'agit uniquement de propriétés de sûreté. Quant à la propriété DSP-4, elle fait référence au passé. En absence d'un mécanisme adapté au sein de la logique temporelle, la seule alternative est de recourir à une spécification opérationnelle, en définissant un moniteur. Le moniteur est également décrit en VHDL. Il est composé avec le contrôleur de cache, et la preuve porte sur les invariants qu'il définit. Le code VHDL de ce moniteur peut être consulté dans l'annexe A.1 de ce document.

Le port de commande Les propriétés liées à cet partie du contrôleur de cache concernent exclusivement la correction des transferts vis-à-vis du protocole de communication attendu. Une manière élégante de spécifier formellement les propriétés inhérentes à ce protocole est suggérée par [76] et consiste également à utiliser le mécanisme des moniteurs. La figure 4.14 présente un graphe d'état très facile à coder en VHDL et qui permet de spécifier et de capturer les incohérences qui peuvent se produire sur l'interface du port de commande.

L'état *inactif* du moniteur traduit l'absence d'activité sur ce port. Lorsqu'une transaction est initiée, le signal *req* est activé. Selon le sens demandé de la transaction, le moniteur se place dans un des états *lecture en cours* ou *écriture en cours*. Chacun de ces états ne

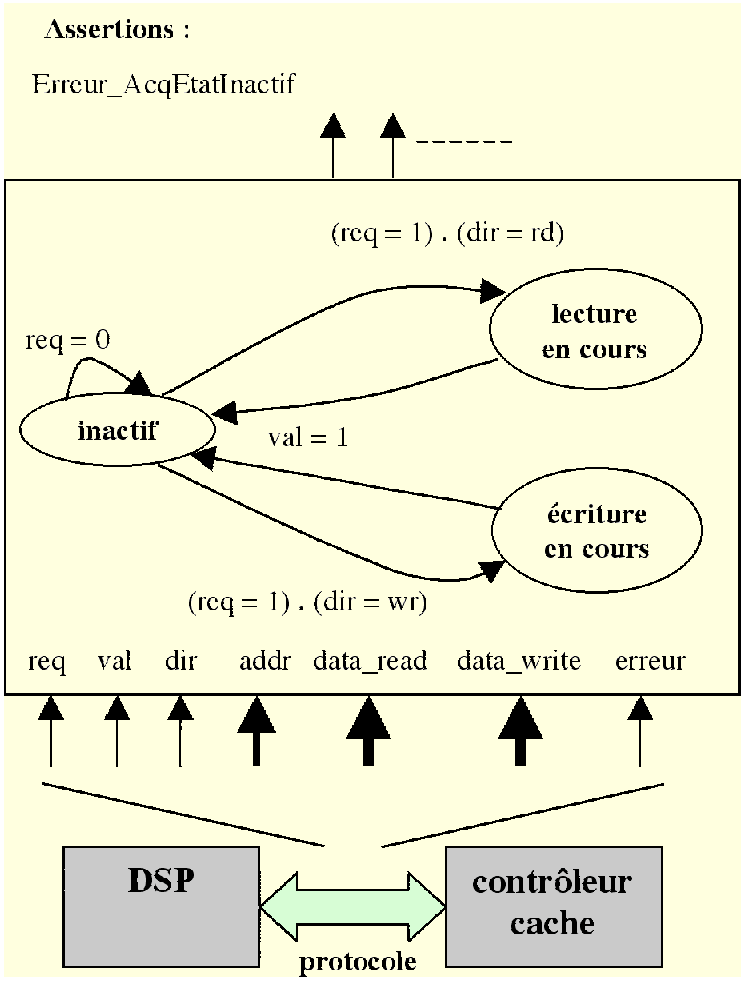

FIG. 4.14 – Moniteur permettant de spécifier le protocole de communication sur le port de commande

peut être quitté que lorsque la transaction est terminée par un acquittement (le signal *val* est activé).

L'assertion étiquetée *Erreur_ AcqEtatInactif* (que l'on abrégera PC-1) exprime le fait qu'une erreur se produit lorsque le signal d'acquittement (*val*) est activé sans requête préalable. Le code VHDL complet de ce moniteur peut être consulté en annexe A.2 de ce document.

Le moteur DMA Voici les aspects que nous retenons pour la vérification de cette partie du contrôleur :

– (DMA-1) *partage équitable du port de chargement externe* : lorsqu'un transfert DMA est en cours, les requêtes du DSP vers la mémoire externe continuent à être acheminées ;

– (DMA-2) *terminaison du transfert* : lorsqu'un transfert DMA est initié, il sera terminé. La page demandée est chargée et le bloc de mémoire cache qui vient d'être raffraichi est validé.

Chacune de ces propriétés se décrit facilement à l'aide de la logique temporelle linéaire ou arborescente. La propriété DMA-1 est une propriété de sûreté alors que la propriété DMA-2 exprime la vivacité.

Le registre d'état interne Ce registre est constitué d'un ensemble de variables d'état, mémorisant chacune un aspect du fonctionnement du contrôleur. Ces variables sont mises à jour par le DSP, à travers des transactions d'écriture spécifiques à travers le port de contrôle. Les seules propriétés intéressantes dans ce contexte spécifient les transactions à utiliser par le DSP (mot d'adresse et mot de donnée) pour mettre à jour chaque élément du registre. Nous retenons deux aspects de fonctionnement de ce bloc :

– (RE-1) *activation du bloc de chargement DSP* : sur le port de commande du contrôleur, l'écriture du mot binaire équivalent à 5 à l'adresse binaire équivalente à 10 a pour effet d'affecter la valeur *actif* au régistre mémorisant l'état actif/inactif du bloc de chargement DSP ;

– (RE-2) *sélection du moteur DMA* : sur le port de commande du contrôleur, l'écriture du mot binaire équivalent à 7 à la même adresse a pour effet d'activer le moteur DMA. De façon consécutive, en spécifiant le numéro du banc de mémoire cache (entre 1 et 8) sur le bus d'adresse, ainsi que l'adresse de base de la page mémoire à télécharger, les paramètres du transfert DMA sont complets et le raffraîchisement d'un banc peut commencer.

4.5.2.4 Vérification par l'approche ascendante

Nous avons écrit des propriétés temporelles en nous limitant, autant que possible, aux entrées-sorties du contrôleur de cache. Cependant, en raison de la taille de ce circuit, la preuve de la majeure partie de ces propriétés ne donne pas de résultats en temps utile, car elle va s'appuyer sur une représentation symbolique très complexe. C'est le cas des propriétés DSP-1,2,4,5 ainsi que DMA-1,2. Ce phénomène est causé par l'absence de hiérarchie dans la description du contrôleur de cache.

Une façon efficace d'aborder ce problème est de forcer une décomposition malgré l'absence de hiérarchie, à travers l'identification manuelle d'un ensemble de *blocs fonctionnels*. C'est une application de l'approche ascendante, évoquée au chapitre précédent. L'analyse du code VHDL permet d'isoler les différents blocs fonctionnels du contrôleur de cache (Figure 4.13). Il est à présent possible de diriger la vérification successivement et indépendamment sur chacun de ces blocs.

Nous allons illustrer cette approche sur la vérification du bloc de chargement DSP. La stratégie que nous présentons a été appliquée de manière analogue sur les autres blocs fonctionnels du contrôleur de cache.

Vérification du bloc de chargement DSP L'essentiel du fonctionnement de ce bloc est conditionné par la valeur du signal VHDL *activation_ DSP*, illustré sur la figure 4.13, qui permet de connaître son mode de fonctionnement, actif ou inactif, tel qu'il a été enregistré

dans le registre d'état interne. Ainsi, la preuve des propriétés DSP-1,2,3,4,5 se fait sur un modèle symbolique qui doit contenir nécessairement un sous-ensemble pertinent du registre d'état interne. Tous les éléments du registre d'état interne sont à leur tour contrôlés par la logique combinatoire et séquentielle associée au port de commande. L'exécution de la vérification montre que le modèle symbolique associé aux propriétés DSP-1,2,3,4,5 est bien trop complexe pour permettre leur évaluation en un temps raisonnable.

Dans le but de simplifier le modèle symbolique associé à ces propriétés, nous allons forcer une étape de décomposition : les propriétés DSP-1,2,3,4,5 concernent le bloc de chargement DSP. On souhaite donc que leur modèle symbolique soit restreint à ce bloc. Pour atteindre cet objectif, il est nécessaire de mettre en œuvre une démarche d'abstraction des traitements associés au registre d'état interne ainsi qu'au port de commande. On crée une frontière ouverte entre le bloc de chargement DSP et son environnement, en considérant le signal *activation_ DSP* comme une *entrée primaire* de ce bloc. Les propriétés sont donc prouvées sur le bloc de chargement DSP **coupé de son entourage**. Cette coupure est effectuée manuellement : on indique à l'outil de preuve d'ignorer les fonctions de transition associées au signal *activation_ DSP* entrant dans ce bloc.

L'interprétation des résultats de la vérification est la suivante. Si une propriété est vraie à l'échelle locale du bloc de chargement DSP, ce résultat a été obtenu sous l'hypothèse que **tous** les comportements possibles peuvent se produire sur le signal *activation_ DSP*. C'est pourquoi la propriété reste vraie dans l'implémentation d'origine.

En revanche, si une parmi les propriétés exprimées est fausse dans le modèle réduit, aucune réponse n'est disponible à l'échelle globale. Dans cette situation il est nécessaire de recourir aux étapes auxiliaires suivantes :

1. valider le contre-exemple fourni par l'outil de vérification à l'échelle du modèle global, à l'aide d'un outil de simulation. Si la même séquence de stimuli produit les mêmes réponses dans le modèle global, alors la propriété est fausse dans le modèle global ;
2. si le contre-exemple ne peut pas être reproduit par le modèle global du contrôleur, cela signifie que l'abstraction choisie est trop grossière. C'est notre cas. En considérant que tous les comportements sont possibles dans l'environnement du bloc de chargement DSP, on accepte également l'éventualité que le signal *activation_ DSP* puisse être inactif à l'infini. Ce scénario est envisageable mais il n'est pas réaliste.
   Une abstraction plus convenable consiste à prendre en compte uniquement les instants pendant lesquels le bloc de chargement DSP est actif, en faisant l'hypothèse que le signal *activation_ DSP* est toujours actif (G (*activation_ DSP*) en logique temporelle). Ce scénario peut être mis en place grâce à la partition comportementale. Le signal *activation_ DSP* étant devenu une entrée primaire du bloc de chargement DSP, il est contraint avec la valeur constante $'1'$. Cette abstraction est très efficace, et permet de vérifier l'ensemble des propriétés DSP-1,2,4,5 d'une manière à la fois rapide et peu coûteuse en mémoire (la propriété DSP-3 n'a pas de sens lorsque le bloc de chargement DSP est actif pour toujours). Cependant, le cas où le bloc de chargement DSP est désactivé en plein milieu du chargement d'un mot d'instruction ne peut pas être vérifié. Une abstraction plus proche de la réalité permettrait de modéliser l'environnement comme une succession de fenêtres temporelles actif-inactif de longueurs "non négligeables", en faisant l'hypothèse que l'activation ou la désactivation du bloc

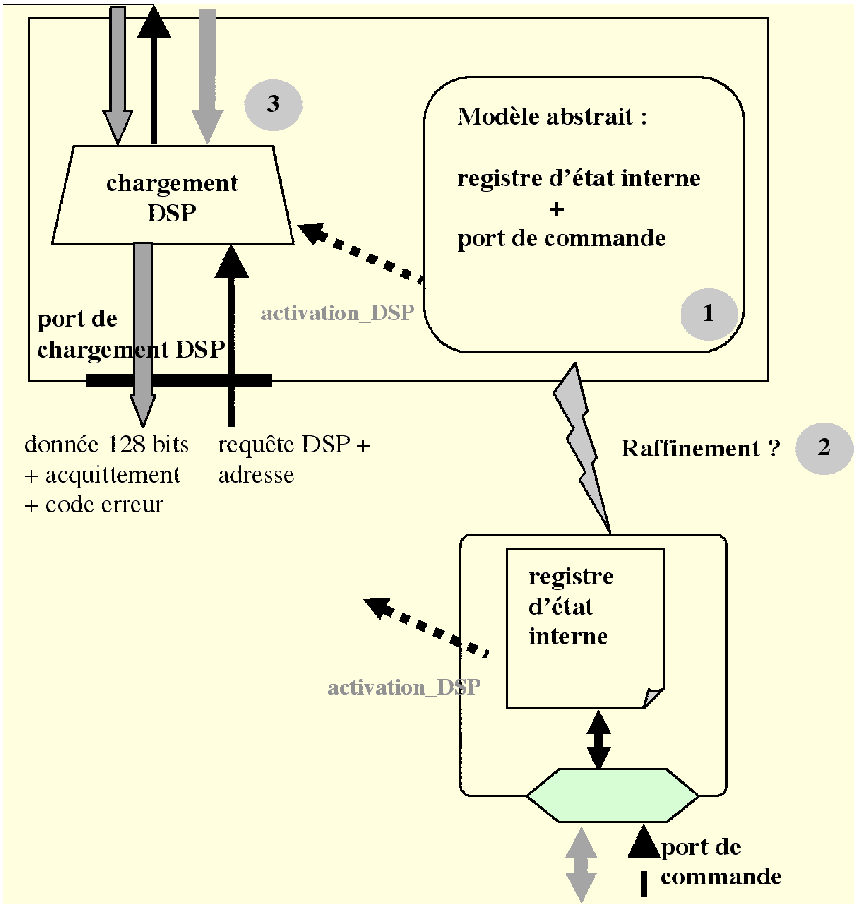

FIG. 4.15 – Raisonnement compositionnel appliqué à la vérification du bloc de chargement DSP

de chargement DSP durent au moins deux cycles d'horloge.

Ce processus doit être réitéré jusqu'à ce que l'outil de vérification affirme que la propriété est vraie ou jusqu'à ce qu'il trouve un contre-exemple satisfaisant (qui met en évidence une erreur dans le circuit ou dans la spécification).

L'écriture d'une abstraction pour le signal *activation_ DSP* est synonyme de mise en place d'une spécification opérationnelle abstraite pour son comportement. Ainsi, en prouvant les propriétés DSP-1,2,3,4,5 relatives au bloc de chargement DSP, sous l'hypothèse de la modélisation abstraite du signal *activation_ DSP*, nous avons effectué une étape du raisonnement "je suppose-tu garantis". Il reste à prouver que la définition abstraite de ce signal peut reproduire l'ensemble des comportements qui peuvent avoir lieu au sein de sa description initiale. En d'autres termes, il est nécessaire d'effectuer une *preuve de raffinement*.

Les étapes du raisonnement "je suppose-tu garantis" appliqué à la vérification du bloc de chargement DSP sont illustrées par la figure 4.15. L'ordre judicieux d'application de ces étapes de raisonnement est le suivant :

Propriété	Taille modèle (variables d'état)	Preuve par décomposition						Preuve directe (sans décomposition)		
		Preuve raffinement			Preuve propriété					
		Mo	BDD	tps	Mo	BDD	temps	Mo	BDD	tps
DSP-1	120	32	$7.5\ 10^5$	2.6	99	$3\ 10^6$	10	238	$7.5\ 10^6$	17
DSP-2	121	"	"	"	192	$8.3\ 10^6$	43	388	$18\ 10^6$	48
DSP-3	121	"	"	"	45	$1.8\ 10^6$	2.6	258	$10.5\ 10^6$	31
DSP-4	128	"	"	"	63	$2.8\ 10^6$	8	400	-	30
DSP-5	165	"	"	"	91	$3.6\ 10^6$	8	400	-	20

TAB. 4.1 – Chiffres comparatifs illustrant l'efficacité d'une démarche compositionnelle

1. écriture d'un modèle abstrait pour décrire le comportement du signal *activation_DSP* ;
2. preuve que le modèle abstrait écrit inclut tous les comportements spécifiés pour *activation_DSP* dans l'implémentation ;
3. preuve des propriétés DSP-1,2,3,4,5 à l'échelle du bloc de chargement DSP composé avec le modèle abstrait de *activation_DSP*.

En effet, la preuve de raffinement doit précéder la preuve des propriétés, car il est nécessaire de s'assurer que le modèle abstrait choisi est valide. L'ensemble des propriétés exprimées sur ce bloc, excepté DSP-3, sont vérifiées. Pour DSP-3, l'outil de vérification fournit un contre-exemple satisfaisant, mettant en évidence une erreur dans le codage du circuit.

Chacune des cinq propriétés ont été vérifiées selon l'approche compositionnelle, en forçant la décomposition et en remplaçant l'environnement du bloc de chargement DSP par une description plus abstraite. L'abstraction que nous avons choisie pour le signal *activation_DSP* permet de définir des fenêtres temporelles arbitraires de longueurs supérieures ou égales à deux cycles d'horloge. La description SMV de ce modèle est donnée dans l'annexe B.1. Ensuite, ces mêmes propriétés ont été évaluées de manière directe, sans aucune tentative de décomposition. Les chiffres présentés dans le tableau 4.1 montrent l'efficacité de la démarche compositionnelle par rapport à l'approche directe. Les temps d'exécution representés sont donnés en minutes. Les colonnes intitulées "BDD" présentent les nombres de nœuds BDD utilisés lors de chaque preuve. La preuve des propriétés DSP-4 et DSP-5 par l'approche directe n'aboutit pas. Nous avons décidé d'arrêter l'exécution lorsque la mémoire utilisée dépasse 400 Méga-octets.

Une manipulation préliminaire nécessaire a été de réduire la taille des bancs de mémoire cache à une valeur raisonnable, permettant d'effectuer le calcul des états atteignables en un temps acceptable. A travers la définition d'une nouvelle abstraction très simple, nous avons donc remplacé la valeur initiale 1024 par 4, sans modifier le source VHDL.

La vérification du contrôleur de cache a été effectuée sur le site de ST-Microelectronics, à l'aide de l'outil industriel Cadence FormalCheck. Cependant, afin de pouvoir bénéficier de la fonctionnalité de preuve de raffinement, qui n'est pas encore implémentée au sein des outils industriels FormalCheck et RuleBase, qui nous sont familiers, nous avons utilisé l'outil Cadence SMV. Soulignons tout de même que les outils FormalCheck et RuleBase permettent également de remplacer une partie d'un circuit par une définition plus abstraite. La preuve a été exécutée sur une architecture Sparc bi-processeur, disposant d'une mémoire

vive de 1,6 Giga-octets et fonctionnant à une cadence de 400 MHz.

La comparaison des performances des outils SMV, du domaine public, et FormalCheck n'est qu'approximative, car ces deux outils ont été utilisés sur des sites différents et sur des ordinateurs différents. Soulignons seulement que ces deux outils, appliqués sur le même circuit, ont des performances comparables : dans les deux cas, les temps de réponse et l'utilisation des ressources restent la plupart du temps acceptables.

Retour sur l'exploitation de la symétrie. La propriété DSP-5 est exprimée sur un objet vectoriel comportant 128 bits. Cet objet représente le port de sortie d'un mot d'instruction vers le DSP. Lorsque le mot d'instruction demandé par le DSP se trouve dans la mémoire cache, le contrôleur lit un mot de 128 bits sur son interface avec les bancs de mémoire, et achemine ce mot vers le DSP. En échange, lorsque le transfert concerne un mot d'instruction stocké dans la mémoire externe, ce mot est transféré à travers un bus de données dont la largeur n'est que de 32 bits. Le mot d'instruction est donc reconstitué après quatre transferts externes.

Une simplification immédiate consiste à décomposer ce but de preuve en 128 sous-buts, à évaluer séparément. Cette opération est vitale pour l'obtention d'une réponse dans un délai raisonnable.

On remarque toutefois qu'en faisant abstraction des transferts externes, les mots d'instruction acheminés entre la mémoire cache et le DSP sont symétriques. Il est donc possible de prouver la correction *des transferts internes* en effectuant un raisonnement sur la propriété de symétrie du port d'instruction. Ainsi, il suffit de prouver qu'un parmi ses éléments constituants est acheminé correctement. Dans le tableau 4.1, on a prouvé la propriété DSP-5 sur le composant d'indice 0 de ce tableau.

L'approche compositionnelle que nous venons d'illustrer a été appliquée d'une manière similaire sur les autres blocs fonctionnels du contrôleur de cache. Elle s'est montrée d'une grande utilité dans ce contexte, où le circuit vérifié n'a pas de hiérarchie. Cependant, pour mettre en œuvre cette approche, l'ingénieur de vérification a besoin d'un certain nombre de détails sur l'implémentation du circuit qu'il souhaite vérifier. La modélisation abstraite du signal *activation_DSP* est le fruit d'une extraction manuelle, basée à la fois sur le document informel de spécification et sur le code source VHDL du contrôleur (le processus VHDL qui affecte la valeur de ce signal s'étend sur cinq pages).

Nous avons mis en œuvre plusieurs modélisations abstraites, notamment en utilisant le non-déterminisme, à travers l'emploi de pseudo-entrées. Elles sont présentées dans l'annexe B.2. Contrairement au cas détaillé ci-dessus, les comportements d'activation et de désactivation du bloc de chargement DSP sont modélisés et vérifiés séparément, au sein de deux modèles distincts. Les propriétés DSP-1,2,3,4,5 ont été évaluées dans les deux contextes induits par ces deux modèles abstraits. Cependant, les performances *globales* de vérification sont légèrement inférieures à celles présentées dans le tableau 4.1. Cette dégradation est expliquée par l'évaluation répétée de chaque propriété sur deux modèles symboliques très proches l'un de l'autre.

La partition fonctionnelle permet de remplacer dans certains cas la décomposition hiérarchique pour vérifier les propriétés d'un circuit. Son application est illustrée sur le même circuit dans la suite de ce document.

4.5.2.5 Vérification par application de la partition fonctionnelle

La stratégie de vérification que nous allons illustrer a pour but d'effectuer une décomposition non pas structurelle mais fonctionnelle sur le modèle du contrôleur de cache. La méthodologie de base que nous allons suivre est la suivante :

- mise en place du mode opératoire souhaité. Ceci se traduit par l'activation de certains blocs fonctionnels du circuit. Cette étape est effectuée par simulation symbolique, avec les vecteurs de simulation appropriés pour une transaction sur le port de commande ;
- inhibition des transactions sur le port de commande, dans le but de figer le circuit dans le mode opératoire qui vient d'être obtenu. Cette opération correspond au pas $k+1$ de l'algorithme 3.1 (page 96) ;
- preuve que le mode opératoire souhaité a effectivement été atteint ;
- vérification des propriétés DSP-1,2,3,4,5, DMA-1,2 et RE-1,2 sur l'état initial du modèle réduit, obtenu par simulation symbolique.

L'expérience montre que la plupart des modes opératoires intéressants pour la vérification du contrôleur de cache ont pu être identifiés grâce au document informel de spécification de ce circuit. Ce document décrit les séquences de simulation nécessaires pour activer ou inactiver un mode opératoire.

En revanche, la correction de cette démarche compositionnelle serait compromise dès que la séquence de simulation symbolique n'est pas explicitement donnée par un document de spécification. En effet, inventer des modes opératoires présente le risque de contraindre de manière excessive le comportement du circuit, et en conséquence de rendre invisible une erreur de conception. La mise en place d'un mode opératoire est donc confrontée aux mêmes limitations que la spécification correcte d'environnement.

Mise en place des modes opératoires sur le contrôleur de cache

La séquence de remise à zéro concerne l'intégralité du circuit. Le signal *reset* est activé pendant un temps fini, puis il est désactivé pour toujours.

La séquence d'activation du bloc de chargement DSP peut être initiée à tout moment ultérieur à la remise à zéro du circuit. Seuls les ports d'entrée appartenant au *port de commande* (au nombre de 60) sont pilotés par des valeurs numériques. Les autres ports du contrôleur de cache (au nombre de 240) restent symboliques. Donc, le *port de commande* doit recevoir une transaction composée des événements suivants :

1. une requête active, accompagnée d'une adresse valide et d'un mot de commande. L'adresse et le mot de commande doivent correspondre à l'entrée du registre d'état interne permettant de modifier le signal *activation_ DSP* ;
2. le contrôleur de cache acquitte cette requête au cycle d'horloge suivant. Les transactions qui écrivent le registre d'état interne sont toujours acquittées en temps constant. A ce moment, le bloc de chargement DSP est actif ;
3. afin de nous assurer que ce bloc reste actif, les transactions ultérieures sur le port de commande sont désactivées. Nous faisons donc le choix de vérifier les fonctionnalités du contrôleur de cache une fois qu'elles ont été activées et sous l'hypothèse qu'elles restent actives à l'infini.

La dernière étape énoncée ci-dessus peut être exprimée sous d'autres variantes moins contraignantes. Il est donc techniquement possible de continuer à autoriser toutes les transactions possibles, ou bien d'autoriser toutes celles qui n'ont pas pour but de désactiver le bloc de chargement DSP, etc. Typiquement, une fois que des résultats satisfaisants ont été obtenus avec des hypothèses très contraignantes sur le modèle, on recommence avec des hypothèses moins restrictives.

La séquence d'activation du bloc DMA nécessite un comportement similaire de la part de l'environnement du contrôleur. Elle a pour effet de démarrer le téléchargement d'une page mémoire dans un des bancs de la mémoire cache. Il s'agit d'enchaîner deux transactions sur le port de commande : l'une spécifie l'adresse de base de la page mémoire à télécharger ainsi que le banc destinataire dans la mémoire cache et l'autre démarre le chargement.

L'ensemble des comportements présentés ci-dessus peuvent être combinés de plusieurs façons, de manière à engendrer certains modes opératoires du contrôleur. Nous illustrons deux combinaisons possibles, permettant d'engendrer les modes opératoires suivants :

- (OP1) effectuer une séquence de remise à zéro, suivie par une séquence d'activation du bloc de chargement DSP. Les propriétés suivantes doivent être vérifiées :
  - RE-1 : le bloc de chargement DSP est actif ;
  - DSP-1,2,4,5. La propriété DSP-3 n'est pas pertinente pour le mode opératoire atteint, car elle concerne le cas où le bloc de chargement DSP est inactif.
- (OP2) activer OP1. Ensuite, demander le chargement d'une page mémoire externe quelconque, spécifier le banc mémoire destinataire du chargement et lancer l'opération DMA. Le bloc de chargement DSP et le moteur DMA doivent tourner en parallèle. Les propriétés DSP-1,2,4,5, RE-1 doivent continuer à être vérifiées, ainsi que les suivantes :
  - RE-2 : sélection du moteur DMA et démarrage du transfert ;
  - DMA-1,2 : le téléchargement se termine en un temps fini, et partage, si nécessaire le port de chargement externe avec les éventuelles requêtes en provenance du DSP.

La même manipulation qui vise à réduire la taille des bancs de mémoire cache est nécessaire dans ce contexte.

La mise en place des modes opératoires présentés ainsi que la vérification des propriétés ont été effectuées à l'aide de l'outil NuSMV. Les résultats chiffrés obtenus ne sont donc pas directement comparables à ceux obtenus par l'approche ascendante. Le tableau 4.2 illustre donc les performances de cette approche, comparée à l'utilisation directe sans aucune démarche compositionnelle. Les temps d'exécution sont donnés en minutes. L'ensemble de ces tests ont été effectués au sein du même environnement (matériel, système d'exploitation) que celui décrit dans la section précédente.

La propriété RE-2 n'est pas pertinente dans le mode opératoire OP1. Elle exprime le démarrage du chargement DMA alors que l'on sait que la séquence nécessaire pour ce démarrage n'a pas eu lieu. De même, les propriétés DMA-1,2 ne sont pas pertinentes dans le mode opératoire OP1.

Les propriétés RE-1 et RE-2 ont été exécutées sur le modèle global du contrôleur, à l'aide d'une modélisation classique des séquences de simulation nécessaires pour mettre en place le mode opératoire souhaité : une spécification opérationnelle d'environnement

Propriété	Preuve par partition fonctionnelle						Preuve directe (sans décomposition)		
	OP1			OP2					
	Mo	BDD	tps	Mo	BDD	temps	Mo	BDD	tps
RE-1	18	$4.7\ 10^5$	0.2	45	$8\ 10^5$	1.5	400	-	-
RE-2	-	-	-	22	$6\ 10^5$	0.3	400	-	-
DSP-1	26	$6.5\ 10^5$	0.5	54	$1.1\ 10^6$	2.2	245	$8.2\ 10^6$	24
DSP-2	18	$5\ 10^5$	1.5	20	$6\ 10^5$	2	392	$20\ 10^6$	48
DSP-4	48	$2\ 10^6$	6	67	$3\ 10^6$	10	400	-	35
DSP-5	102	$4.5\ 10^6$	14.5	110	$4.8\ 10^6$	18	400	-	23
DMA-1	-	-	-	12	$5\ 10^5$	2	400	-	150
DMA-2	-	-	-	16	$6\ 10^5$	1.5	400	-	120

TAB. 4.2 – Chiffres comparatifs illustrant l'efficacité de la partition fonctionnelle

(machine d'états). Cette machine d'états doit être composée avec le modèle du contrôleur avant de démarrer la vérification. Le modèle produit est trop complexe et donc inutilisable dans l'état.

L'évaluation de l'ensemble des propriétés présentées s'est appuyé sur les options suivantes :

- réduction par cône d'influence structurel activée ;
- évaluation des formules CTL grâce à l'algorithme de parcours en arrière, y compris pour la propriété DSP-4 ;
- calcul préalable des états atteignables désactivé, *sauf* pour les propriétés DMA-1,2, pour lesquelles cette étape préliminaire apporte une amélioration considérable. Les temps d'exécution pour ces deux propriétés tiennent compte du calcul des états atteignables. Pour le reste des propriétés, l'emploi du calcul des états atteignables n'entraîne pas d'amélioration notable.

Dans ce contexte, l'emploi du parcours "en avant", adapté à la preuve des invariants, ne donne pas de résultats significativement différents. L'essentiel de la réduction est obtenu grâce à la partition fonctionnelle.

Nous avons également tenté de refaire la vérification des propriétés DSP-1,2,4,5, RE-1,2 et DMA-1,2 en utilisant une séquence de simulation symbolique légèrement différente. Afin de nous situer dans les mêmes conditions que la preuve compositionnelle abordée au paragraphe précédent, le dernier ($k + 1^{\text{ème}}$) pas de simulation symbolique est beaucoup moins restrictif et permet à toutes les transactions possibles d'avoir lieu sur le port de commande. Il a été surprenant de constater que dans l'ensemble, les performances de vérification sont légèrement supérieures cette fois-ci, malgré le fait que les expressions du modèle booléen sous-jacent devraient à priori être plus complexes.

Bien qu'il soit difficile d'expliquer ce phénomène sur un modèle aussi complexe, une explication probable serait la suivante. La taille du BDD qui représente une fonction caractéristique de n variables dépend du cardinal de l'ensemble qu'il doit représenter, mais aussi des éléments contenus dans cet ensemble. En règle générale, les ensembles dont le cardinal est proche de 0 ou de 2^n sont représentés de manière compacte. Par ailleurs, si les monômes qui représentent les éléments de ces ensembles sont très différents les uns des

autres (ce qui se traduit par une distance de Hamming importante), alors cette tendance est inversée aux environs des cardinaux bas. Il est donc raisonnable de supposer que la relation de transition du nouveau modèle réduit permet de coder un ensemble de combinaisons de cardinal important et/ou dont les éléments peuvent être représentés de manière compacte. Il est bien sûr très difficile d'accéder à ce type d'information avant la vérification du circuit. Mais cette remarque souligne la grande importance du choix du codage des états pour l'efficacité de la représentation symbolique.

Conclusion et perspectives

Les approches automatisées de conception et de vérification sont aujourd'hui matures, lorsque la description de départ ne dépasse pas le niveau d'abstraction "transfert de registres". Cependant, au sein d'un projet, la vérification reste l'activité prépondérante. Le choix d'une technique de vérification est fortement dépendant de la nature du circuit à vérifier, ainsi que des propriétés à évaluer. Dans le contexte actuel, la simulation traditionnelle, guidée par l'utilisateur ou aléatoire (guidée par des contraintes), constitue encore l'outil le plus répandu et le plus productif.

La vérification de modèles est présente dans l'industrie sous diverses formes plus ou moins flexibles et puissantes, et s'appuie sur diverses techniques de représentation et de preuve. Elle vient seulement compléter la démarche centrale de simulation, lorsque celle-ci échoue : lors de la preuve de propriétés de sûreté ou de vivacité dans des systèmes critiques.

Bien que très similaire à la synthèse comportementale, l'extraction d'un modèle booléen pour la vérification de modèles doit tenir compte de règles bien particulières pour la modélisation correcte et efficace des comportements mémorisants et combinatoires. Nous avons étudié la mise en œuvre de ces aspects au sein de deux outils industriels de vérification de modèles, ainsi que sur notre propre outil d'extraction, V2SMV.

Deux approches d'extraction sont courament utilisées par les environnements industriels. L'outil RuleBase d'IBM s'appuie sur des logiciels externes pour effectuer cette étape. Par exemple, une étape de synthèse à l'aide de l'environnement de conception Synopsys permet d'engendrer un réseau de portes logiques et de bascules, décrit en VHDL, et lisible directement par l'outil de vérification. Quant à l'outil FormalCheck de Cadence, il s'est appuyé jusqu'à ce jour sur son propre utilitaire d'extraction.

L'emploi d'un outil de synthèse pour extraire un modèle booléen offre une grande fiabilité d'extraction. Cependant, contrairement à ce que l'on serait tenté de croire, une étape de synthèse et de minimisation de la logique combinatoire n'implique pas d'amélioration notable sur l'efficacité de la vérification symbolique. Le seul gain est obtenu en début de preuve, lors de la construction des BDDs qui représentent le modèle symbolique.

Par contraste, un outil spécifique d'extraction de modèles booléens tel que celui de FormalCheck permet de conserver des informations concernant la hiérarchie, qui sont vitales lors d'une démarche compositionnelle, et qui apportent aussi plus de commodité pour l'utilisateur. En revanche, même si cet outil couvre un sous-ensemble largement suffisant du langage VHDL de niveau RTL, on constate que sa mise en œuvre n'est pas aussi efficace que celle d'un outil dédié à la synthèse.

Le choix que nous avons fait pour extraire un modèle booléen à partir de VHDL a été de conserver également la hiérarchie au sein du modèle extrait afin de favoriser les démarches compositionnelles. Nous offrons actuellement la possibilité de compiler un sous-ensemble VHDL de niveau RTL synthétisable vers le format d'entrée de plusieurs outils de vérification gratuits : VIS, Cadence-SMV et NuSMV. Nous ne proposons pas d'étape de minimisation de la logique combinatoire, car une telle étape est superflue dans le contexte de la vérification symbolique basée sur les BDDs. Quant à la logique séquentielle, notre outil génère un ensemble très compact de variables d'état. Son efficacité de ce point de vue est identique à celle des outils industriels de synthèse, à un point près : les outils industriels éclatent les objets vectoriels en bits ; c'est pourquoi la propriété de mémorisation est associée de manière indépendante à chacun des éléments du vecteur. Par contraste, au sein de notre outil, tous les éléments d'un vecteur de bits ont la même propriété de mémorisation.

La combinaison de la simulation symbolique et de la vérification de modèles a fait l'objet de travaux déroulés en parallèle avec les nôtres [48]. Hazelhurst et al. proposent d'utiliser la technique STE [47] afin d'exécuter toute séquence initiale requise au fonctionnement normal d'un circuit. Une étape de vérification de modèles suit. L'inconvénient de leur approche vient des représentations symboliques incompatibles de ces deux techniques. Le résultat de la simulation symbolique a besoin d'être converti d'une représentation paramétrique en une représentation symbolique avant de pouvoir démarrer la preuve. Cette conversion est une opération coûteuse en général [48].

Notre mise en œuvre intégre naturellement la simulation symbolique au sein d'un outil de vérification, en utilisant uniquement les mécanismes de base des outils de vérification, et ne nécessite pas d'étape de conversion préliminaire. L'intégration d'une telle fonctionnalité au sein des outils industriels de vérification de modèles présenterait de très grands avantages. En effet, la plupart des outils offrent déjà la possibilité de modéliser des comportements "en entrée" sous la forme de séquences explicites dont le suffixe est constant et de longueur infinie. Ces comportements sont traduits en une machine d'états qui est automatiquement composée avec le circuit à vérifier, ce qui complique dès le départ son modèle symbolique. Par contraste, une étape de simulation symbolique permet de mettre en œuvre la même modélisation d'environnement tout en réduisant le modèle symbolique de deux manières : en s'avançant de quelques pas dans l'espace d'états du modèle et en simplifiant les expressions booléennes grâce au mécanisme du co-facteur.

Les outils industriels RuleBase d'IBM et FormalCheck de Cadence, que nous avons utilisés, ont fait l'objet d'une étude initiale approfondie, dont l'objectif était de faire un choix d'acquisition. En ce qui concerne les outils universitaires, nous avons essentiellement utilisé Cadence SMV pour sa fonctionnalité de preuve de raffinement, et NuSMV, au sein duquel nous avons intégré l'outil de simulation symbolique.

Actuellement, les travaux de recherche en matière de vérification de circuits s'orientent massivement vers l'utilisation des techniques SAT, en remplacement des BDDs pour la vérification de modèles, sur un nombre de pas d'exécution borné ou non. Des travaux très récents illustrent l'efficacité de cette technique [65]. Nous avons acquis la conviction que la complexité des circuits industriels conservera une longueur d'avance par rapport aux progrès technologiques en matière de représentation de leur modèle et d'algorithmes de

preuve. Malgré les progrès importants dans le domaine des algorithmes fondamentaux, les méthodes de vérification exposées dans ce document conserveront une importance vitale dans la maîtrise de la complexité du circuit vérifié.

Perspectives. Nous avons présenté une stratégie nouvelle de vérification, basée sur la simulation symbolique. Cette stratégie utilise des séquences initiales de longueur finie, permettant de décomposer le comportement du système vérifié. Cette approche peut être étendue aux séquences de longueur infinie, dans le cadre d'une approche de vérification axée sur un parcours en avant. En effet, compte-tenu du fait que la relation de transition extraite d'une description VHDL est totale, les traces commençant à l'état initial sont de longueur infinie et possèdent un suffixe périodique. Il s'agit d'identifier de tels comportements périodiques et de les utiliser pour réduire le modèle à la volée lors du parcours en avant. Un tel comportement est caractéristique du signal d'horloge, ou encore du signal de remise à zéro.

La preuve exhaustive des circuits décrits au niveau RTL se heurte rapidement au problème d'explosion combinatoire. Il est donc intéressant de vérifier la plupart des propriétés du circuit au niveau de sa spécification abstraite, afin de raisonner sur un modèle comportant moins d'éléments, puis de prouver l'existence d'une relation de raffinement entre la spécification et l'implémentation de niveau RTL.

Une direction importante de recherche met l'accent sur la spécification opérationnelle de haut niveau du circuit. Des extensions de langages existants ont engendré les normes SystemC [70] et System-Verilog [3]. Des travaux sont également en cours pour rendre possible la spécification de haut niveau à l'aide du langage Java. Toutefois, le passage d'une spécification de haut niveau vers une description RTL synthétisable ne bénéficie pas encore d'une démarche systématique et conservative pour la correction du circuit. Par ailleurs, comme pour VHDL, ces langages doivent être restreints à des sous-ensembles adaptés pour la vérification formelle.

Les techniques automatiques de réduction constituent, elles aussi, une appproche prometteuse dans un contexte industriel. Elles sont basées sur des heuristiques de décomposition structurelle du modèle booléen du circuit. Plusieurs mises en œuvre existent à ce jour et ont prouvé leur efficacité au sein d'outils tels que FormalCheck. Une telle étape de décomposition intervient typiquement avant la construction des BDDs du modèle symbolique pour la preuve. Il serait intéressant de mener une étude comparative des performances des approches structurelles existantes par rapport à l'approche du cône de réduction fonctionnel, présentée dans ce document.

Enfin, une direction très intéressante est celle d'une approche de synthèse, utilisant les méthodes formelles pour engendrer des descriptions correctes par construction à partir d'une spécification logique. Cette approche a atteint aujourd'hui un certain niveau de maturité, mais reste toutefois en retrait par rapport à la vérification de modèles.

Annexe A

Description VHDL des moniteurs utilisés

A.1 Moniteur permettant de vérifier la propriété DSP-4

```
library IEEE;
use ieee.std_logic_1164.all;
use work.c2vhdl.all;

entity PIMIRecognizer is
  port(
    pMEM_p_dr: in STD_LOGIC_VECTOR(127 downto 0);
    pMEM_p_adr: in STD_LOGIC_VECTOR(27 downto 0);
    pMEM_p_rq: in STD_LOGIC;
    pMEM_p_val: in STD_LOGIC;
    pMEM_p_default_grant: in STD_LOGIC;
    pMEM_p_err: in STD_LOGIC_VECTOR(2 downto 0);

    -- Error outputs :
    ErrValOnlyWhenReq : out boolean;

    clk_clk: in STD_LOGIC;
    rst_rst: in STD_LOGIC
    );
end PIMIRecognizer;

architecture a of PIMIRecognizer is
  signal r1,r2,r3 : std_logic;
  signal a1,a2,a3 : std_logic_vector(27 downto 0);
  signal int, extern : boolean;

  signal int_q : boolean;
  signal ext_q : boolean;

begin  -- a
```

```
  r1 <= pMEM_p_rq when pMEM_p_default_grant = '1' else '0';
  a1 <= pMEM_p_adr;

  process (pMEM_p_rq,pMEM_p_adr,pMEM_p_default_grant)
  begin  -- process
    extern <= false;
      if (pMEM_p_rq = '1') and (pMEM_p_default_grant = '1') then
        if pMEM_p_adr(27 downto 20) = "00000000" then
          if (pMEM_p_adr(19 downto 10) >= "0100000000") then
            extern <= true;
          end if;
        end if;
      end if;
  end process;

  req_s: process (clk_clk, rst_rst)
  begin  -- process req_s
    if rst_rst = '0' then               -- asynchronous reset (active low)
      r2 <= '0';
      r3 <= '0';
      a2 <= "00000000000000000000000000000000";
      a3 <= "00000000000000000000000000000000";
      ext_q <= false;

    elsif clk_clk'event and clk_clk = '1' then  -- rising clock edge
      if extern then
        ext_q <= true;
      end if;
      r2 <= r1;
      a2 <= a1;
      if ext_q and (pMEM_p_val= '1') and (r3 = '0') and (not extern) then
        ext_q <= false;
      end if;
      r3 <= r2;
      a3 <= a2;
    end if;
  end process req_s;

  -- Signal sortant caractérisant une erreur (DSP-4):
  --
  ErrValOnlyWhenReq <=  (pMEM_p_val = '1') and ((r3 = '0') and not ext_q);
end a;
```

A.2 Moniteur permettant de vérifier les propriété de correction du port de commande

```
library WORK;
use WORK.Recognizers_pack.all;
library IEEE;
use ieee.std_logic_1164.all;

entity RV_recognizer is
    generic (MaxAddrBit, MinAddrBit, MaxDataBit, MinDataBit: in integer);
    port
    (req: in STD_LOGIC;
     addr: in STD_LOGIC_VECTOR(MaxAddrBit downto MinAddrBit);
     be: in STD_LOGIC_VECTOR(3 downto 0);
     rwb: in STD_LOGIC;
     val: in STD_LOGIC;
     err: in STD_LOGIC;
     dataWrite: in STD_LOGIC_VECTOR(MaxDataBit downto MinDataBit);
     dataRead: in STD_LOGIC_VECTOR(MaxDataBit downto MinDataBit);
     dataWriteVar: in STD_LOGIC_VECTOR(MaxDataBit downto MinDataBit);
     dataReadVar: in STD_LOGIC_VECTOR(MaxDataBit downto MinDataBit);
     addrVar: in STD_LOGIC_VECTOR(MaxAddrBit downto MinAddrBit);
     beVar: in STD_LOGIC_VECTOR(3 downto 0);
     errVar: in STD_LOGIC;
     clock : in STD_LOGIC;
     Resetn : in STD_LOGIC;
--   Ports signaux d'erreur (PC-1)
     Err_ValInIdle: out boolean;
-- .......
    );

  type state_type is (idle, readInProgress, writeInProgress);  -- control
                                                               -- states
end RV_recognizer;

-- architecture RTL synthetizable de la spécification du port de contrôle
architecture rtl of RV_recognizer is

signal  state: state_type := idle ;

-- Memorisation des valeurs précédentes de addr et de dataWrite
signal  prevAddr:  STD_LOGIC_VECTOR(MaxAddrBit downto MinAddrBit);
signal  prevDataWrite: STD_LOGIC_VECTOR(MaxDataBit downto MinDataBit);
signal  prevBe:  STD_LOGIC_VECTOR(3 downto 0);
```

```
begin
-- delayed values from addr and dataWrite bus
clocked: process(clock)
  begin
    if clock'event and clock='1' then
        if Resetn = '0' then
            state <= idle;
            prevBe <= "0000";
        else
            prevAddr <= addr;
            prevDataWrite <= dataWrite;
            prevBe <= be;
            case state is                              -- graphe d'état
                when idle => if req = '0' then
                                 state <= idle;
                             elsif rwb = '1' then
                                 state <= readInProgress;
                             else
                                 state <= writeInProgress;
                             end if;
                when readInProgress | writeInProgress => if val = '1' then
                                                            state <= idle;
                                                         end if;
                when others => state <= state;
            end case;
        end if;
    end if;
end process clocked;

-- constraint / error conditions of RV protocol, relative to control states
Err_ValInIdle <= (state = idle) and  (val = '1');
                                  when (state = readInProgress)
                                  else false ;
-- .........
end rtl;
```

Annexe B

Modélisation abstraite

B.1 Modèle abstrait utilisé pour le signal *activation_DSP*

Le modèle abstrait que nous avons choisi de mettre en œuvre est décrit à l'aide de la version Cadence du langage SMV :

```
layer fetch_stall_model : {
init(pimi.DSP_FETCH_STALL) := 0;
 if(pimi.IFB_REG_WE & reset)
   if((PIFB_PIMI_RV_ADR[3..0] = [1,0,1,0]) &
      (PIFB_PIMI_RV_DW[4..0] = [0,0,1,0,0]) &
      pimi.CLK_CLK_rising_edge)
     next(pimi.activation_DSP) := 0;
   else if((PIFB_PIMI_RV_ADR[3..0] = [1,0,1,0]) &
           (PIFB_PIMI_RV_DW[4..0] = [0,0,1,0,1]) &
           pimi.CLK_CLK_rising_edge)
     next(pimi.activation_DSP) := 1;
}
```

B.2 Modèle abstrait décomposé utilisé pour le signal *activation_DSP*

Le modèle abstrait *fetch_stall_model* peut être décomposé en deux modèles distincts, en utilisant les affectations non-déterministes ($\{0,1\}$), ainsi qu'en exprimant séparément les comportements d'activation :

```
layer fetch_stall: {
init(pimi.activation_DSP) := 0;
 if(pimi.activation_DSP = 1)
   if(pimi.IFB_REG_WE & reset & (PIFB_PIMI_RV_ADR[3..0] = [1,0,1,0]) &
      (PIFB_PIMI_RV_DW[4..0] = [0,0,1,0,0]) &
       pimi.CLK_CLK_rising_edge)
      next(pimi.activation_DSP) := 0;
   else next(pimi.activation_DSP) := pimi.activation_DSP;
 else  next(pimi.activation_DSP) := {0,1};
}
```

et de désactivation :

```
layer fetch_unstall : {
init(pimi.activation_DSP) := 0;
 if(pimi.activation_DSP = 0)
  if(pimi.IFB_REG_WE & reset & (PIFB_PIMI_RV_ADR[3..0] = [1,0,1,0]) &
     (PIFB_PIMI_RV_DW[4..0] = [0,0,1,0,1]) &
      pimi.CLK_CLK_rising_edge)
     next(pimi.activation_DSP) := 1;
  else next(pimi.activation_DSP) := pimi.activation_DSP;
 else  next(pimi.activation_DSP) := {0,1};
}
```

Bibliographie

[1] IEEE Computer Society W.G. 1076.6. *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*. IEEE, 2000.

[2] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on SAT-solvers. In Susanne Graf and Michael Schwartzbach, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems TACAS, Berlin, Germany*, volume 1785. Springer-Verlag, 2000.

[3] Accelera Organization, Inc. *SystemVerilog 3.1 Accellera's Extensions to Verilog*, 2003.

[4] Accellera EDA Standards organization, Napa, CA, USA. *SUGAR Formal Specification Language Reference Manual - DRAFT*, May 2002.

[5] R. Airiau, J.-M. Bergé, V. Olive, and J. Rouillard. *VHDL - du langage à la modélisation*. Presses Polytechniques et Universitaires Romandes, 1990.

[6] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27 :509–516, June 1978.

[7] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA : Modularity in model checking. In *Computer Aided Verification*, pages 521–525, 1998.

[8] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. *Lecture Notes in Computer Science*, 1536 :81–102, 1998.

[9] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579 :193–207, 1999.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[11] Roderick Bloem, In-Ho Moon, Kavita Ravi, and Fabio Somenzi. Approximations for fixpoint computations in symbolic model checking. In *Systemics, Cybernetics and Informatics*, 2000.

[12] Roderick Bloem, Kavita Ravi, and Fabio Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *Computer Aided Verification*, pages 222–235, 1999.

[13] Roderick Bloem, Kavita Ravi, and Fabio Somenzi. Symbolic guided search for CTL model checking. In *Proceedings of the Design Automation Conference*, pages 29–34, Los Angeles, CA, 2000. ACM.

[14] D. Borrione, M. Boubekeur, and E. Dumitrescu et al. An approach to the introduction of formal validation in an asynchronous circuit design flow. In Jr. R. H. Sprague, editor, *Proceedings of the* 36^{th} *Annual Hawaii International Conference on System Sciences (HICSS'03)*. IEEE Computer Society, 2003.

[15] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[16] Gianpiero Cabodi, Paolo Camurati, Luciano Lavagno, and Stefano Quer. Disjunctive partitioning and partial iterative squaring : An effective approach for symbolic traversal of large circuits. In *Design Automation Conference*, pages 728–733, 1997.

[17] Cadence Design Systems, Inc., San Jose, USA. *FormalCheck User Guide*, 2001.

[18] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV : A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4) :410–425, 2000.

[19] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2) :244–263, 1986.

[20] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2) :77–104, 1996.

[21] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[22] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs : Workshop*, number 131 in LNCS, New-York, 1981. Springer.

[23] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications : A practical approach. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 117–126, Austin, 1983.

[24] C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[25] F. Copty, L. Fix, E. Giunchiglia, and M. Vardi G. Kamhi, A. Tacchella. Benifits of bounded model checking at an industrial setting. In *Proceedings of CAV*, LNCS, pages 436–453. Springer, 2001.

[26] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Springer, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373, 1990.

[27] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 299–310, Standford, California, USA, 1994. Springer-Verlag.

[28] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear time temporal logic. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 249–260. Springer-Verlag, Berlin, 1999. LNCS 1633.

[29] A. Debreil and P. Oddo. Synchronous designs in VHDL. In *Proceedings of the Europeean Design Automation Conference (EURO-DAC)*, pages 486–491. IEEE CS Press, 1993.

[30] D. Deharbe. *Vérification formelle de propriétés temporelles : étude et application au langage VHDL*. PhD thesis, Université Joseph Fourier - Grenoble 1, 1996.

[31] D. Déharbe. Model checking on finite state machines : extensions and applications to VHDL designs. In *Proceedings of the First Asian-Pacific Conference on Hardware Description Languages : Standards and Applications*, Brisbane, Australia, 1993.

[32] E. Dumitrescu. Identification d'éléments mémorisants dans des descriptions VHDL synchrones. Master's thesis, Université Joseph Fourier, 1998.

[33] Emil Dumitrescu. *System-on-Chip for Real-Time Applications*, chapter "A Practical Approach to the Formal Verification of SoC's with Symbolic Model Checking", pages 98–110. Kluwer, 2002.

[34] Emil Dumitrescu and Dominique Borrione. Symbolic Simulation as a Simplifying Strategy for SOC Verification with Symbolic Model Checking. In *Proceedings of IWSOC 2003*. CS Press, July 2003. to appear.

[35] Emil Dumitrescu and Pierre Ostier. Identification of non-redundant memorizing elements in VHDL synchronous designs for formal verification tools. In *Proceedings of the Fifth International Workshop on Symbolic Methods and Applications in Circuit Design (SMACD)*, pages 233–243, 1998.

[36] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design : An International Journal*, 9(1/2) :105–131, August 1996.

[37] E. Encrenaz. *Une méthode de vérification de propriétés de programmes VHDL basée sur des modèles formels de réseaux de Petri*. PhD thesis, Université Pierre et Marie Curie, Paris IV, 1995.

[38] Hans Eveking. *Architecture Validation and Verification Methods*, chapter Machine Assisted Verification. Lipari, Italy, 1997.

[39] Kathi Fisler and Moshe Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In *Formal Methods in Computer-Aided Design*, pages 115–132, 1998.

[40] Kathi Fisler and Moshe Y. Vardi. Bisimulation and model checking. In *Conference on Correct Hardware Design and Verification Methods*, pages 338–341, 1999.

[41] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[42] A. Gill. Introduction to the theory of finite-state machines. In McGraw Hill, editor, *Electronic Science Series*, 1962.

[43] E. Goldberg and Y. Novikov. Berkmin : A fast and robust sat solver. In *Proceedings of DATE*, pages 142–149, 2002.

[44] S. Graf. *Logique du temps arborescent pour la spécification et la preuve de programmes*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 1984.

[45] The VIS Group. Vis : A system for verification and synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification*, LNCS, pages 428–432. Springer, 1996.

[46] O. Grumberg and D.E. Long. Model checking and modular verification. In J. C. M. Baeten and J. F. Groote, editors, *Proceedings of CONCUR'91 : 2nd International Conference on Concurrency Theory*, volume 527 of *LNCS*. Springer, 1991.

[47] S. Hazelhurst and C. J. Seger. Symbolic trajectory evaluation. In T. Kropf, editor, *Formal Hardware Verification : Methods and Systems in Comparison*, number 1287 in LNCS, pages 3–79, Berlin, 1997. Springer-Verlag.

[48] S. Hazelhurst and O. Weissberg. A hybrid verification approach : Getting deep into the design. In *Proceedings of DAC*, pages 111–116, New orleans, LA, June 2002.

[49] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee : Methodology and case studies. In *Computer Aided Verification*, pages 440–451, 1998.

[50] Thomas A. Henzinger, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. An assume-guarantee rule for checking simulation. In *Formal Methods in Computer-Aided Design*, pages 421–432, 1998.

[51] S. Höreth. A word-level graph manipulation package. *Software Tools for Technology Transfers*, 3(2) :182–192, 2001.

[52] K. A. Sakallah J. P. M. Silva. GRASP : A Search Algorithm for Propositional Satisfiability. *IEEE Transactions of Computers*, 48 :506–521, 1999.

[53] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, 1991. North-Holland.

[54] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4) :401–424, 1994.

[55] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking : 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[56] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw Hill, 1978.

[57] Yuji Kukimoto. Blif-mv. Technical report, The VIS Group. University of California, Berkeley, 1996.

[58] Orna Kupferman and Moshe Y. Vardi. Freedom, weakness, and determinism : From linear-time to branching-time. In *Proceedings 13th IEEE Symposium on Logic in Computer Science*, pages 81–92, Juin 1998.

[59] IBM Haifa Research Laboratories. *RuleBase User's Guide, v1.4.4*, February 2003.

[60] LEDA. *LEDA VHDL System User's Manual.* France, 1996.

[61] David Long. *Model checking, abstraction, and compositional verification.* PhD thesis, Carnegie Mellon University, 1993.

[62] K. McMillan. Getting started with SMV, 1998. Cadence Berkeley Labs. Available on the web page http ://www.cis.ksu.edu/santos/smv-doc/tutorial/tutorial.html.

[63] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[64] K. L. McMillan. Verification of infinite state systems by compositional model checking. In L. Pierre and T. Kropf, editors, *Proceedings of CHARME*, number 1703 in LNCS, pages 219–233, Germany, 1999. Springer.

[65] K. L. McMillan. Interpolation and sat-based model-checking. In Hunt and Somenzi, editors, *Proceedings of CAV'03*, number LNCS 2725 in LNCS, pages 1–13. Springer, 2003.

[66] In-Ho Moon, James H. Kukula, Kavita Ravi, and Fabio Somenzi. To split or to conjoin : the question in image computation. In *Design Automation Conference*, pages 23–28, 2000.

[67] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[68] D.E. Muller, A. Saoudi, and P. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, pages 422–427, 1988.

[69] J.C. Madre O. Coudert, C. Berthet. Verification of Synchronous Sequential Machines based on Symbolic Execution. In J. Sifakis, editor, *Workshop on Automatic Verification Methods for Finite State Systems*, number 407 in LNCS, pages 365–373. Springer Verlag, June 1989.

[70] Open SystemC Initiative. *SystemC 2.0.1 Language Reference Manual*, 2003.

[71] A. Pardo and G. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In O. Grumberg, editor, *9th Int. Conference on Computer Aided Verification (CAV'97)*, pages 12–23. Springer-Verlag, June 1997. LNCS-1254.

[72] Abelardo Pardo and Gary D. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In *Computer Aided Verification*, pages 12–23, 1997.

[73] A. Pnueli. *Logics and Models of Concurrent Systems*, chapter In transition from global to modular temporal reasoning about programs, pages 123–144. Springer-Verlag New York, Inc., 1985.

[74] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *Proceedings of IEEE/ACM International Workshop on Logic Synthesis*, Lake Tahoe, USA, May 1995.

[75] K. Ravi and F. Somenzi. Hints to accelerate symbolic traversal. In L. Pierre and T. Kropf, editors, *Proceedings of CHARME'99*, volume 1703 of *LNCS*, pages 250–264, 1999.

[76] Kanna Shimizu, David L. Dill, and Alan J. Hu. Monitor-based formal specification of pci. In *Proceedings of Formal Methods in Computer-Aided Design*, November 2000. The PCI specification is available on the web page http ://radish.stanford.edu/pci.

[77] IEEE Computer Society. *IEEE Standard VHDL Language Reference Manual*, 2000.

[78] IEEE Computer Society. *IEEE 1364 Verilog Language Reference Manual*, 2001.

[79] F. Somenzi. CUDD : CU Decision Diagram Package Release, 1998.

[80] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 248–263. Springer-Verlag, Berlin, July 2000.

[81] G. Stålmark. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Technical report, Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995), 1989.

[82] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language, Fifth Edition*. Kluwer, 2002.

[83] P. Williams, A. Biere, E. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In E.A. Emerson and A.P. Sistla, editors, *Proceedings of CAV*, LNCS 1855, pages 124–138. Springer-Verlag, July 2000.

[84] Hantao Zhang. SATO : an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.

Résumé

La vérification symbolique de systèmes matériels est limitée par la complexité exponentielle en taille de représentation du modèle symbolique sous-jacent. Ce travail porte sur la réduction, manuelle ou non, de ce modèle. Les approches compositionnelles structurelles et comportementales ont été étudiées dans un contexte industriel. Cette étude a précédé le développement d'une nouvelle technique de réduction : la partition fonctionnelle. Cette technique s'applique aux systèmes dont le comportement est décomposable séquentiellement. La partition fonctionnelle est mise en place grâce à une étape préliminaire de simulation symbolique. Elle a été implémentée et appliquée sur un circuit industriel de taille importante, et a permis d'obtenir d'excellents résultats en matière de réduction. L'expérimentation des techniques de preuve présentées s'est appuyée sur un outil d'extraction de machines d'états finis à partir de descriptions VHDL qu'il a été nécessaire de mettre en œuvre.

Abstract

Symbolic model checking applied to hardware designs is limited by the exponential complexity in the size of the underlying verified model. This work explores several issues for achieving model-reduction, either manually or on an automated basis. The structural and behavioral approaches of compositional verification have been studied in an industrial design context. This study has enabled the development of a new model reduction technique : the functional partitioning. This technique is intended for those systems whose behavior can be sequentially decomposed. It relies on a preliminary symbolic simulation phase which is performed prior to the actual verification. Functional partitioning has been implemented and applied on an industrial design of non-trivial size and brings spectacular model-reduction improvements. In support of the experiments presented here, a VHDL to finite state machines translator tool has also been developed and used.

ISBN 2-84813-021-0 (Broché)
ISBN 2-84813-022-9 (Version électronique)