Vérification formelle des systèmes numériques par démonstration de théorèmes: application aux composants cryptographiques

D. Toma

UNIVERSITÉ JOSEPH FOURIER (GRENOBLE I)

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Spécialité : INFORMATIQUE

École Doctorale Mathématiques Informatique Sciences et Technologies de l'Information

présentée et soutenue publiquement
par

Diana Toma

le 18 juillet 2006

Vérification Formelle des systèmes numériques par démonstration de théorèmes : application aux composants cryptographiques

---

JURY

Rapporteurs	M. Marc DAUMAS	Chargé de recherches CNRS, LIRMM
	M. Claude KIRCHNER	Directeur de recherches INRIA, LORIA Nancy
Examinateurs	Mme. Boutheina CHETALI	Directeur de recherche, Gemalto
	Mme. Laurence PIERRE	Professeur, Univ. de Nice-Sophia Antipolis
Directeur de thèse	Mme. Dominique BORRIONE	Professeur, Univ. Joseph Fourier

Thèse préparée au sein du Laboratoire TIMA
"Techniques de l'Informatique et de la Microélectronique
pour l'Architecture des Ordinateurs"

Remerciements

Je souhaite tout d'abord remercier les membres du jury, qui ont permis la soutenance de ma thèse :

- Marc Daumas et Claude Kirchner, pour avoir accepté la tâche de rédiger les rapports de soutenance. Merci pour leur relecture approfondie du manuscrit, leurs suggestions et leurs encouragements.
- Boutheina Chetali et Laurence Pierre pour m'avoir fait l'honneur de faire partie du jury de thèse. Merci à Laurence Pierre pour ses remarques lors de la préparation de la soutenance.

Je voudrais exprimer ma gratitude à ma directrice de thèse, Dominique Borrione, qui m'a encadrée et soutenue pendant toutes ces années. Ses remarques et ses encouragements m'ont toujours fait avancer dans la vie de chercheur. Un grand merci pour m'avoir donné la possibilité d'aller à de nombreuses conférences et d'avoir soutenu mon séjour dans le groupe ACL2. Dominique a été non seulement un mentor, mais aussi une amie dont les conseils ont été précieux.

Mes remerciements vont également à Solomon Neculai pour m'avoir initiée aux mystères des mathématiques ; à Gabriel Ciobanu pour m'avoir donné le goût de la recherche et m'avoir soutenue dans mes démarches pour passer quelques mois en France ; à Traian Muntean qui m'a convaincue et encouragée à poursuivre ma recherche en France, étape qui a été ensuite si décisive pour mon avenir. Je remercie Warren Hunt pour m'avoir invitée à passer quelques mois au sein du group ACL2, à l'Université du Texas, et m'avoir ainsi donné la possibilité d'étendre mes connaissances sur ACL2 et les Etats Unis. Aussi, grand merci à Jean Mermet de m'avoir fait rentrer dans la vie professionnelle alors que ma rédaction de thèse n'était pas achevée, cette dernière m'a paru ainsi moins laborieuse.

Je souhaite remercier tous les membres de l'équipe VDS. En particulier Eric Gascard, ami et soutien indispensable pour ses conseils judicieux et ses encouragements, et surtout sa disponibilité tellement appréciable pour moi. Je n'oublie pas les échanges très enrichissants avec Ghiath Al Sammane, tant sur le plan scientifique que culturel, ainsi que les passionnantes discutions avec Menouer Boubekeur autour du café. J'ai beaucoup apprécié l'affection paternelle que m'a apportée Claude Le Faou, et la dose d'humour quotidienne d'Emil Dumitrescu et Pierre Ostier. Je vous remercie vous tous et vos conjoints pour votre amitié sincère, et pour les nombreux moments inoubliables que nous avons passés ensemble. Je tiens également à exprimer ma sympathie à tous les autres membres de l'équipe VDS et du laboratoire TIMA que j'ai eu le plaisir de rencontrer.

Un grand merci à Benedicte Fluxa pour sa patience et son aide dans la correction de ce manuscrit.

Toute ma gratitude également aux membres du groupe ACL2 qui m'ont accueillie pendant mon séjour avec beaucoup de chaleur et d'amitié. Merci surtout à Warren, Anna, Serita et Hangbing qui m'ont accueillie comme un véritable membre de leur famille.

Enfin, je remercie mes parents et ma famille qui m'ont toujours soutenue dans mes choix, même si la séparation et l'éloignement du milieu familial a été douloureuse pour tous.

Je ne remercierai jamais assez mon mari Bogdan d'être resté à mes côtés et d'avoir accepté de me suivre, même en pays inconnu, afin que je puisse réaliser cette thèse.

Je termine par une pensée toute particulière à mes amis pour leur soutien moral et affectif qui comblent ma vie.

Table des matières

Liste des figures

Liste des tableaux

Introduction générale

Un processus de conception typique d'un système sur puce (SoC[1]) débute avec une spécification qui définit la fonctionnalité du système à concevoir. La spécification est donnée à l'aide d'un langage naturel, à l'aide d'un langage de programmation, à l'aide d'un langage de description matérielle ou bien à l'aide d'autres formalismes de spécification comme UML. Par la suite, cette spécification est raffinée manuellement ou automatiquement afin d'obtenir une description de plus bas niveau.

Dans le flot de conception industriel, l'ensemble de fonctionnalités du système conduit à un modèle d'architecture, implémenté comme un simulateur en C, C++ ou SystemC. Le modèle est ensuite raffiné et testé pour sa correction fonctionnelle. Si des erreurs sont trouvées, le modèle plus abstrait est modifié et le processus de raffinement suivi de test recommence. Quand le modèle est satisfaisant, le code correspondant au niveau transfert de registre en VHDL ou Verilog (les deux langages industriels dominants) est produit. La traduction du modèle C en RTL est un processus manuel et implique l'introduction de plusieurs détails, comme la communication au niveau cycle. Il existe des outils de synthèse comportementale mais le sous-ensemble du HDL pris en compte est très réduit, et le résultat de la synthèse, n'étant pas optimal, ne correspond pas encore aux attentes des concepteurs. Le modèle RTL sert d'entrée pour un outil de synthèse logique comme Synopsys's Design Compiler, Synplicity's Synplifier, ou Mentor Graphics's LeonardoSpectrum qui le transforme dans un circuit au niveau portes. Le circuit est l'entrée d'un système de placement et routage qui génère le dessin de masques.

Afin de garantir la correction du flot de conception, il est necessaire de :
- valider la spécification initiale.
- vérifier que le passage d'un niveau d'abstraction à l'autre est correct.

A cause de la complexité croissante des SoC, la vérification devient un aspect très important : 70-80% du coût de conception est alloué à cette tâche. Selon Blank [14] plus de 60% des projets de développement d'ASIC doivent être repris à cause des erreurs fonctionnelles. Généralement, environ 50% des erreurs de conception sont situées au niveau du module. Dans le monde industriel, malheureusement, la vérification est trop souvent

[1] System on Chip

synonyme de simulation. Pour vérifier un pas de conception, une spécification et son implémentation sont simulées pour les mêmes entrées afin d'obtenir les mêmes sorties. Les erreurs de spécification et d'implémentation ainsi détectées sont corrigées. Le processus de simulation est repris jusqu'à ce que des erreurs ne se manifestent plus. Cette démarche ne garantit pas l'absence d'erreurs.

Même si la simulation est une méthode de vérification naturelle pour les concepteurs, elle a de nombreux inconvénients. Ces inconvénients sont liés à la qualité des tests : ils doivent couvrir entièrement le comportement du circuit, de façon à mettre en évidence les erreurs. Les tests doivent être également compréhensibles et assez courts puisque la simulation est un processus très coûteux en temps de calcul. Une autre difficulté à la vérification par la simulation est la reconnaissance des erreurs mises en évidence lors de la simulation et l'estimation de la complétude des batteries de test. Ces problèmes restent encore des questions ouvertes dans le domaine de la méthodologie de vérification par la simulation. Bentley [8] a rapporté que pour la vérification du Pentium 4, plus de 200 milliards de cycles ont été simulés, correspondant à environ 2 minutes CPU, pour une fréquence de 1 GHz. Puisque le test exhaustif n'est pas possible, la validation par simulation est une méthode limitée.

Une alternative est fournie par la vérification formelle qui prouve mathématiquement qu'un circuit satisfait une spécification, au lieu d'observer les traces d'exécution et chasser les erreurs. L'expérience a démontré que l'utilisation des méthodes formelles mène à une réduction des coûts, car la création de tests n'est plus nécessaire [138]. Des méthodes hybrides ont été développées, comme la simulation symbolique ou la vérification par assertions. Ces méthodes, appelées semi-formelles, font le lien entre la simulation et la vérification formelle.

Les techniques de vérification formelle peuvent être classifiées en deux catégories :

- des méthodes automatiques, basées sur la vérification de modèles,
- des méthodes déductives basées sur la démonstration de théorèmes.

Ces méthodes peuvent s'appliquer à tout type de système : séquentiel ou combinatoire, synchrone ou asynchrone, de type contrôle ou de type chemin de données, et à différents niveaux d'abstraction : à haut niveau (algorithmique, transfert de registres) ou bas niveau (portes, transistors).

La vérification de modèles est mise en pratique de deux manières : l'équivalence de modèles (*equivalence checking*), et la vérification temporelle de modèles (*model checking*).

L'équivalence de modèles vérifie si deux descriptions matérielles sont équivalentes du point de vue fonctionnel. Cette technique peut être appliquée au niveau du transfert de registres et/ou au niveau des portes, le niveau d'abstraction des deux descriptions n'étant pas nécessairement le même. L'utilisation de l'équivalence de modèles permet de vérifier soit l'étape de synthèse logique, soit une étape d'optimisation. Les outils sont automatiques et sont capables de traiter aujourd'hui des conceptions avec plusieurs millions de portes.

La vérification temporelle de modèle vérifie si une description matérielle respecte une propriété temporelle. Cette technique est appliquée au niveau transfert de registres. Les

formules de la logique temporelle expriment le comportement des systèmes dans le temps et sont spécifiées dans un langage de vérification comme PSL, OpenVera, SystemVerilog, etc.

La vérification de modèle utilise comme modèle sémantique le modèle booléen de la machine d'états finis extrait à partir de la description matérielle. Une problèmatique de l'utilisation des méthodes basées sur les machines d'états finis (FSM[2]) est le problème de l'explosion d'états : le nombre d'états augmente exponentiellement avec le nombre des variables d'état de la description. Même si plusieurs techniques (comme le parcours symbolique ou les méthodes structurales) ont été introduites afin de résoudre ce problème, avec la croissance en complexité et en taille des circuits, le modèle booléen est insuffisant. De plus, bien que la logique temporelle soit très adaptée pour décrire le comportement dynamique, elle est inadaptée pour décrire des propriétés algorithmiques (en effet, seule la logique propositionnelle est prise en compte). En conséquence, la vérification de modèle est en général applicable pour le niveau transferts de registre et le niveau portes. Par contre, elle n'est pas capable de répondre à la problématique de vérification pour les niveaux d'abstraction plus hauts.

Les méthodes déductives basées sur la démonstration de théorèmes utilisent soit la logique de premier ordre, soit la logique d'ordre supérieur. La relation entre la spécification et l'implémentation est un théorème à démontrer dans une logique. L'implémentation fournit les axiomes et les hypothèses pour la preuve. La démonstration est basée sur la déduction logique, la réécriture et le mécanisme de la preuve par récurrence. Cette technique est très puissante pour les descriptions matérielles de haut niveau et des systèmes réguliers ou très complexes, car la taille de données n'a plus d'importance. Elle s'applique à tous les niveaux d'abstraction, et est capable de traiter des spécifications non bornées, ainsi que de prouver des propriétés algorithmiques complexes. Grâce à la récurrence, des propriétés fonctionnelles sur des circuits paramétrés peuvent être vérifiées.

Le premier inconvénient de cette technique est la nécessité que l'utilisateur guide le système pour la preuve. Cet inconvénient peut être palié par la bonne connaissance de l'outil de preuve. Le second est lié au fait que la modélisation de la description matérielle se fait dans la logique choisie, ce qui rend l'accès difficile pour les concepteurs. Dans la plupart des approches de vérification qui utilisent la démonstration de théorèmes, les systèmes matériels sont décrits directement avec des concepts de la logique. Une alternative est alors d'utiliser une autre théorie pour caractériser les circuits et implémenter la théorie dans la logique.

Notre travail a comme but de faciliter l'introduction des outils de démonstration de théorèmes dans le flot de conception. Pour cela nous avons choisi parmi les langages de description matérielle, VHDL - un des langages standards utilisés dans la conception des circuits. VHDL permet de décrire un système matériel à tous les niveaux d'abstraction.

[2]Finite State Machine

Pour pouvoir raisonner formellement sur des descriptions matérielles VHDL, nous avons été confrontés au problème de modéliser la sémantique VHDL dans un démonstrateur de théorèmes.

Dans cette thématique de recherche, différents travaux existent. Van Tassel [129] a décrit la sémantique de VHDL en HOL, Russinoff [113] et Georgelin [47] ont défini en ACL2 des fonctions sémantiques pour les éléments de la syntaxe VHDL. Leurs approches utilisent les capacités du démonstrateur de théorèmes pour simuler les descriptions matérielles et vérifient des propriétés sur les résultats de la simulation symbolique. Les travaux utilisent des sous-ensembles très restreints de VHDL et la preuve de résultats est fastidieuse étant donné la manière d'enfouir la sémantique dans les logiques.

Nous proposons un passage par un modèle intermédiaire au lieu de traduire la sémantique du langage directement dans le démonstrateur. A partir d'une sémantique opérationnelle de VHDL, nous définissons un ensemble de transformations afin d'obtenir un modèle sémantique basé sur des équations récurrentes par rapport au temps. Pour chaque objet nous définissons une seule équation qui représente le comportement du circuit entre deux points successifs de synchronisation.

Maintenant que nous avons à notre disposition une modélisation du système à vérifier dans l'outil de preuve, il nous reste à établir sa correction fonctionnelle. Nous allons montrer que les résultats fournis par le système sont en concordance avec la spécification initiale.

Nous établissons alors une séparation entre la partie simulation et la partie preuve du système : la simulation est réalisée par un simulateur symbolique externe basé sur des événements, et le résultat est traduit dans la logique d'un démonstrateur au choix. Dans ce cas, la modélisation n'est plus dépendante d'un démonstrateur, l'utilisateur étant libre de choisir son propre outil de raisonnement.

Plusieurs manières de réaliser la simulation sont possibles, afin d'obtenir des visions différentes du circuit. En particulier, pour les systèmes synchronisés par une horloge mère, nous obtenons le comportement pour un cycle d'horloge. Ceci permet de construire un modèle récursif au niveau du cycle d'horloge. Les résultats de la simulation symbolique sont traduits dans un démonstrateur de théorèmes afin de réaliser des preuves sur le comportement du circuit.

Contrairement aux modèles symboliques des machines d'états finis, où l'espace de données doit être fini afin d'être représenté dans le domaine de la logique propositionnelle, nous n'avons pas de telles restrictions. Ainsi, notre modèle prend en compte les types de données infinis comme les entiers, ou des tableaux non bornés, les déclarations paramétrées, les fonctions récursives et les boucles paramétrées.

Notre approche peut être utilisée soit pour vérifier une étape de synthèse, soit pour vérifier un raffinement. L'utilisation d'un démonstrateur de théorèmes permet la vérification de tous les types d'abstractions ainsi que la vérification formelle de la spécification. Par conséquent, le flot de conception est validé dans sa totalité. Le manuscrit décrit cette approche d'introduction de la technique de démonstration de théorèmes dans le flot de conception. Pour valider le modèle proposé, nous avons vérifié des propriétés de conformité et d'équivalence pour une bibliothèque de circuits de cryptographie en utilisant le

démonstrateur ACL2.

Organisation du mémoire

Le mémoire s'organise de la manière suivante :

- Le premier chapitre présente l'état de l'art des méthodes de vérification formelle.
- Le deuxième chapitre présente le modèle SER basé sur des équations récurrentes, que nous proposons pour la vérification formelle des circuits. Le modèle est au niveau du cycle de simulation. Nous montrons comment obtenir ce modèle à partir d'une description VHDL. Nous présentons en suite la notion de circuit synchrone et nous montrons comment utiliser la simulation symbolique afin d'obtenir un modèle au niveau du cycle d'horloge pour cette catégorie de circuits. La définition du modèle des équations récurrentes est un travail effectué en étroite collaboration avec Ghiath Al Sammane.
- Le troisième chapitre présente le démonstrateur de théorème ACL2 et montre comment le modèle SER est traduit dans la logique du démonstrateur.
- Le quatrième chapitre présente une méthodologie de preuve par récurrence des circuits basée sur le modèle proposé. Nous avons validé l'approche par des études de cas qui ont été prouvées à l'aide du démonstrateur ACL2.
- Dans le dernier chapitre nous présentons nos conclusions et quelques perspectives.

Chapitre 1

Présentation de méthodes et outils de vérification pour les circuits numériques

Il est impossible de savoir si la spécification d'un système est correcte ou complète. Comment savoir si ce que nous avons écrit correspond à nos intentions ? Ainsi, il n'existe pas un système correct dans l'absolu, mais il est possible de vérifier si un système satisfait sa spécification ou non. La vérification formelle implique la construction d'une telle preuve.

La relation de satisfaction est soit une équivalence soit une implication logique. Du point de vue de la vérification, il existe plusieurs critères de correction [63] :

La vérification d'équivalence vérifie que pour les mêmes entrées, l'implémentation et la spécification fournissent les mêmes sorties. Généralement, ce type de vérification est réalisé pour des circuits au même niveau d'abstraction, afin de vérifier une étape d'optimisation.

La vérification de conformité est appliquée pour vérifier une étape de conception. La spécification et l'implémentation sont décrites aux différents niveaux d'abstraction. Melham [89] identifie quatre types d'abstraction utilisés dans la vérification des circuits :

- *l'abstraction structurelle* élimine les détails internes de l'implémentation. La spécification donne une vision 'boîte noire' du circuit en décrivant le comportement observable sans décrire la structure qui le génère.
- *l'abstraction comportementale* élimine les détails sur le comportement du circuit dans les conditions de l'environnement qui ne doivent jamais se produire. Donc le comportement de l'abstraction englobe le comportement du circuit.
- *l'abstraction de données* fait le lien entre les éléments de l'implémentation et les éléments de la spécification quand ils ont des représentations différentes. Par exemple, un signal est représenté comme un entier entre 0 et 5 dans la spécification, et comme un vecteur de 3 bits dans l'implémentation. Pour réaliser une abstraction de données il faut définir une fonction sémantique pour les éléments de l'implémentation dans le domaine sémantique de la spécification.
- *l'abstraction temporelle* fait le lien entre les pas temporels de l'implémentation et les pas temporels de la spécification. Par exemple, une opération réalisée par la spécification dans une unité de temps, est réalisée par l'implémentation dans trois unités. Afin de réaliser une abstraction temporelle, il faut définir une correspondance

entre les échelles de temps des deux descriptions. Ceci n'est pas trivial, car parfois, une unité de temps dans la spécification ne correspond pas à un nombre fixe de pas dans l'implémentation.

La vérification de propriétés logiques et temporelles vérifie la fonctionnalité ou le comportement du circuit dans le temps. Les propriétés sont exprimées dans différentes logiques (de premier ordre, d'ordre supérieur, temporelles).

Des algorithmes et méthodologies ont été développés pour chaque type de vérification, et en conséquence deux types de méthodes se distinguent : la vérification de modèle (model checking) et la vérification basée sur la démonstration de théorèmes (theorem proving).

Pour la vérification de modèle il existe deux types d'outils qui l'implémentent : des vérificateurs d'équivalence, qui prennent en entrée deux descriptions HDL, et des vérificateurs temporels de modèles qui prennent en entrée une description HDL et une propriété temporelle à vérifier.

Les démonstrateurs de théorèmes sont des outils plus généraux qui n'ont pas été conçus pour la vérification de descriptions matérielles. Ils se classifient en fonction de la logique qu'ils utilisent dans des démonstrateurs de premier ordre et des démonstrateurs d'ordre supérieur. La description matérielle et sa spécification doivent être décrites dans la logique du démonstrateur.

Aucun de ces outils n'est capable de traiter la problématique de vérification dans sa totalité. Il faut donc utiliser le bon outil pour la bonne propriété.

Dans ce chapitre, nous présentons les deux méthodes de vérification en détaillant le modèle de l'implémentation, de la spécification et comment différents types de vérification sont réalisés.

1.1 Le modèle de machine à états finis

Nous commençons avec un rappel sur le modèle de la machine à états finis, le modèle formel le plus utilisé pour la représentation d'un circuit (les fondements théoriques sont présentés dans [48]). Par défaut nous considérons uniquement les machines à état déterministes. Les notations suivantes sont inspirées de [42].

Définition 1 (Machine à états finis déterministe) *Une machine à états finis déterministe est un n-uplet*

$$\mathcal{M} = \langle \mathcal{I},\, \mathcal{O},\, \mathcal{S},\, \sigma_0,\, \Delta,\, \Lambda \rangle$$

- $\mathcal{I}$ *est l'ensemble fini de symboles d'entrée ;*
- $\mathcal{O}$ *est l'ensemble fini de symboles de sortie ;*
- $\mathcal{S}$ *est l'ensemble fini d'états ;*
- $\sigma_0 \in \mathcal{S}$ *est appelée l'état initial ;*
- $\Delta : \mathcal{I} \times \mathcal{S} \rightarrow \mathcal{S}$ *est la fonction de transition d'état de* $\mathcal{M}$ *;*
- $\Lambda : \mathcal{I} \times \mathcal{S} \rightarrow \mathcal{O}$ *est sa fonction de sortie.*

La définition est très adaptée pour les circuits synchrones séquentiels. Dans le cas des circuits combinatoires, l'ensemble $\mathcal{S}$ est vide, la fonction de transition n'est pas définie, et la fonction de sortie dépend seulement des entrées.

Dans la littérature il existe deux cas particuliers de machines à états finis :

- *la machine de Mealy* : la fonction de sortie de la machine dépend réellement des entrées et de l'état ;
- *la machine de Moore* : la fonction de sortie de la machine est indépendante des entrées $\Lambda : \mathcal{S} \rightarrow \mathcal{O}$.

Cette classification est pertinente car la tâche de vérification peut être facilitée en sachant que l'interconnexion de deux machines de Moore reste une machine de Moore, mais l'interconnexion de deux machines de Mealy peut introduire des boucles combinatoires, et donc des états instables.

Une machine à états est généralement associée à une échelle discrète de temps. Donc, l'état initial σ_0 de M est associé à l'instant '0', et pour tout instant t, M se trouve dans l'état $\sigma_t \in \mathcal{S}$.

Définition 2 (Configuration courante) *Une configuration d'une machine M à l'instant t est un triplet*

$$C_t = \langle \xi_t, \sigma_t, o_t \rangle$$

ou $\xi_t \in \mathcal{I}$, $\sigma_t \in \mathcal{S}$, $o_t \in \mathcal{O}$ *et* $\Lambda(\xi_t, \sigma_t) = o_t$.

Définition 3 (Etat successeur) *Soit* $\sigma, \sigma' \in \mathcal{S}$ *deux états d'une machine M. L'état* σ' *est successeur de l'état* σ, $\sigma \rightarrow \sigma'$ *si et seulement si* $\exists \xi \in \mathcal{I} : \sigma' = \Delta(\xi, \sigma)$.

Définition 4 (Exécution) *Un exécution d'une machine à états M est une séquence (finie ou non) de configurations* (C_0, C_1, ...) *telle que* $\forall k \in \mathbb{N}$ *ou* $C_k = \langle \xi_k, \sigma_k, o_k \rangle$, $\sigma_{k+1} = \Delta(\xi_k, \sigma_k)$.

1.2 La vérification de modèles

La vérification de modèles est une technique algorithmique pour les systèmes finis qui vérifie par une recherche exhaustive de l'espace d'états du système qu'une propriété donnée est vraie. Le modèle sémantique du système est la machine à états finis. La propriété à prouver est spécifiée soit par une machine à états finis soit par une formule dans la logique temporelle. L'algorithme parcourt les états atteignables du circuit pour vérifier la propriété. Si la propriété n'est pas vraie, un contre-exemple est généré automatiquement sous forme de trace. La recherche se termine toujours dans un temps fini, puisque l'espace d'états est fini.

Pour appliquer cette technique, la machine à états finis est représentée par un modèle booléen. Pour chaque ensemble $\mathcal{I}$, $\mathcal{O}$ et $\mathcal{S}$, un codage booléen est défini : leurs éléments sont associés avec des n-uplets sur $\mathbb{B}$ [37]. Pour que cette correspondance soit réalisée, la taille minimale de n-uplets est :

$$m = \lceil log_2\ card(\mathcal{I}) \rceil \quad p = \lceil log_2\ card(\mathcal{O}) \rceil \quad n = \lceil log_2\ card(\mathcal{S}) \rceil$$

ou *card* calcule les cardinaux des ensembles $\mathcal{I}$, $\mathcal{O}$ et $\mathcal{S}$, et $\lceil x \rceil$ représente le plus petit nombre entier tel que le nombre réel $x \leq \lceil x \rceil$. Donc $\mathcal{I}$, $\mathcal{O}$ et $\mathcal{S}$ deviennent des ensembles de propositions atomiques dans la logique propositionnelle et les fonctions Δ et Λ sont remplacées par leur correspondants binaires. Les valeurs constantes booléennes $\mathbb{B} = \{faux, vrai\}$ de la logique propositionnelle sont associées aux entiers '0' et '1'.

Définition 5 (Modèle booléen de la machine à états finis déterministe) *Soit* $\mathcal{M} = \langle \mathcal{I}, \mathcal{O}, \mathcal{S}, \sigma_0, \Delta, \Lambda \rangle$ *une machine à états finis.* $M = \langle I, O, S, s_0, \delta, \lambda \rangle$ *est une représentation booléenne de* $\mathcal{M}$ *s'il existe trois fonctions d'encodage injectives* $B_I : \mathcal{I} \rightarrow \mathbb{B}^m$, $B_O : \mathcal{O} \rightarrow \mathbb{B}^p$, $B_S : \mathcal{S} \rightarrow \mathbb{B}^n$ *telles que :*

- $s_0 = B_S(\sigma_0)$;
- *la fonction de transition* $\delta : \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}^n$ *vérifie* $\forall \xi \in \mathcal{I}, \forall \sigma \in \mathcal{S}, \delta(\mathcal{B}_\mathcal{I}(\xi), \mathcal{B}_\mathcal{S}(\sigma)) = \mathcal{B}_\mathcal{S}(\Delta(\xi, \sigma))$;
- *la fonction de sortie* $\lambda : \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}^p$ *vérifie* $\forall \xi \in \mathcal{I}, \forall \sigma \in \mathcal{S}, \lambda(\mathcal{B}_\mathcal{I}(\xi), \mathcal{B}_\mathcal{S}(\sigma)) = \mathcal{B}_\mathcal{O}(\Lambda(\xi, \sigma))$;

Les fonctions d'encodage sont injectives mais pas nécessairement surjectives. Dans la plupart des cas, les ensembles I, O et S sont des sous-ensembles stricts de $\mathbb{B}^m$, $\mathbb{B}^p$ et $\mathbb{B}^n$.

Il existe une relation directe entre un circuit et sa représentation par une machine à états booléenne. Les ensembles I et O sont les combinaisons de valeurs des ports d'entrée et de sortie du circuit. Les fonctions de transition et de sortie sont représentées par le réseau de portes du circuit, et les variables d'états sont représentées par les mémoires (bascules). Dans le modèle booléen le temps est discret et toute transition a comme résultat le changement de valeur d'au moins une variable. Dans le circuit physique, ceci correspond à la transition déclenchée par des évènements sur un signal synchronisant, appelé horloge.

Pour des raisons de concision, tout au long de la présentation de la vérification de modèles, nous allons utiliser le terme de machine à états finis pour le model booléen de la machine.

Dans la pratique, il existe deux approches de vérification de modèles. Une première approche est la vérification temporelle des modèles [33]. Dans cette approche les spécifications sont exprimées dans la logique temporelle [7] et les systèmes sont des machines à états finis. Dans la deuxième approche, équivalence de modèles, tant le système que sa spécification sont exprimés par une machine à états finis et l'équivalence entre les deux est vérifiée. Vardi *et al.* [131] montre que le problème de vérification temporelle de modèles peut être transformé en un problème de vérification d'automates, reliant les deux approches.

1.2.1 L'équivalence de modèles

L'équivalence séquentielle vérifie que deux circuits synchrones compatibles (ayant les mêmes entrées et sorties), mais ayant des états différents, produisent des sorties identiques pour la même séquence d'entrées, à partir de leur état initial.

L'équivalence de modèles s'applique aux machines à états finis correspondant aux descriptions matérielles.

Définition 6 (Machines à états compatibles) *Soient* $M_1 = \langle \mathcal{I}_1, \mathcal{O}_1, \mathcal{S}_1, \sigma_{01}, \delta_1, \lambda_1 \rangle$ *et* $M_2 = \langle \mathcal{I}_2, \mathcal{O}_2, \mathcal{S}_2, \sigma_{02}, \delta_2, \lambda_2 \rangle$ *deux machines d'états finis.* M_1 *et* M_2 *sont compatibles si et seulement si elles ont la même interface :* $\mathcal{I}_1 = \mathcal{I}_2$ *et* $\mathcal{O}_1 = \mathcal{O}_2$.

Définition 7 (Etats équivalents) *Deux états* $\sigma_1 \in M_1$ *et* $\sigma_2 \in M_2$ *sont équivalents,* $\sigma_1 \equiv \sigma_2$, *si et seulement si* M_1 *et* M_2 *sont compatibles et* $\forall \xi \in \mathcal{I}_1$, $\lambda_1(\xi, \sigma_1) = \lambda_2(\xi, \sigma_2)$.

Deux machines à états finis sont équivalentes si dans tous les états atteignables, elles produisent les mêmes sorties.

Définition 8 (Equivalence séquentielle) *Deux machines d'états finis* M_1 *et* M_2 *sont séquentiellement équivalentes* $M_1 \equiv M_2$ *si et seulement si :*

- M_1 *et* M_2 *sont compatibles ;*
- *leurs états initiaux sont équivalents :* $\sigma_1 \equiv \sigma_2$ *;*
- $\forall \sigma_1 \in \mathcal{S}_1$ *et* $\sigma_2 \in \mathcal{S}_2$, *si* $\sigma_1 \equiv \sigma_2$ *alors* $\forall \xi \in \mathcal{I}$, $\Delta_1(\xi, \sigma_1) \equiv \Delta_2(\xi, \sigma_2)$.

Pour vérifier l'équivalence de deux machines à états déterministes M_1 et M_2, leur produit M_P est construit.

Définition 9 (Produit de machines à états) *Soient deux machines d'états finis compatibles* $M_1 = \langle I, O, S_1, s_{01}, \delta_1, \lambda_1 \rangle$ *et* $M_2 = \langle I, O, S_2, s_{02}, \delta_2, \lambda_2 \rangle$. *Leur produit* $M_P = \langle I, o, S_P, s_{0P}, \delta_P, \lambda_P \rangle$ *est une machine à états booléens obtenue de la façon suivante :*

- *L'ensemble d'états de la machine produit* M_P *est le produit ensembliste des états :* $S_P = S_1 \times S_2$ *;*
- *L'état initial de* M_P *est la concaténation des états initiaux de* M_1 *et* M_2 *:* $s_0P = s_01.s_02$ *;*
- *La fonction de transition est la concaténation de* δ_1 *et* δ_2 *:* $\delta_P : I \times S_1 \times S_2 \rightarrow S_1 \times S_2$, *et* $\forall \xi \in I$, $\forall s_1 \in S_1$, $\forall s_2 \in S_2$, $\delta_P(\xi, s_1, s_2) = (\delta_1(\xi, s_1), \delta_2(\xi, s_2))$ *;*
- *La fonction de sortie est un ou exclusif des sorties de* M_1 *et* M_2 *:* $\lambda_P : I \times S_1 \times S_2 \rightarrow \mathbb{B}$, *et* $\forall \xi \in I$, $\forall s_1 \in S_1$, $\forall s_2 \in S_2$, $\lambda_P(\xi, s_1, s_2) = \lambda_1(\xi, s_1) \neq \lambda_2(\xi, s_2)$ *;*

Donc, vérifier si deux machines à états sont équivalentes revient à calculer les états atteignables de M_P et vérifier que dans chaque état, pour toutes les valeurs possibles des entrées, la sortie vaut 0. Le parcours de l'espace d'états part de l'état initial (reset) et explore tout chemin d'exécution possible de la machine. Pour tout état atteignable, si la sortie est 0 l'espace d'états est incrémenté, sinon, les descriptions ne sont pas équivalentes. Dans ce cas, un contre-exemple est généré. Puisque l'espace d'états est fini, l'algorithme se termine quand chaque état est atteint au moins une fois.

Si les deux circuits sont combinatoires, la vérification d'équivalence revient à vérifier que, pour toutes les entrées, les sorties des deux circuits sont les mêmes.

Définition 10 (Equivalence combinatoire) *Deux machines finies combinatoires M_1 et M_2 sont équivalentes $M_1 \equiv M_2$ si et seulement si :*

- *M_1 et M_2 sont compatibles ;*
- *$\forall \xi \in \mathcal{I}$, $\lambda_1(\xi) \equiv \lambda_2(\xi)$.*

Les méthodes basées sur des machines à états finis ont le problème connu dans la littérature sous le nom d'explosion d'états : le nombre d'états augmente exponentiellement avec le nombre de variables d'état de la description. Plusieurs techniques ont été introduites afin de résoudre ce problème, comme le parcours symbolique et les méthodes structurales. Le parcours symbolique utilise des structures symboliques comme les OBDD[1] pour représenter la machine à états finis, au lieu d'un graphe explicite. Les ensembles d'états explicites sont remplacés par des fonctions caractéristiques. Les états sont codés par des variables booléennes et représentées par des OBDD. Le parcours explicite de l'espace des états est remplacé par la manipulation fonctionnelle des OBDD. Le parcours basé sur les OBDD ne résoud pas définitivement le problème d'explosion d'états, traitant des machines d'états contenant environ 100 variables d'état. Cette limite est poussée plus loin par les méthodes structurales [79], basées sur la règle 'divide et impera' : le circuit est divisé et des points de coupure sont introduits. Différentes techniques de décision sont appliquées afin de comparer ou simplifier les partitions.

1.2.2 La vérification temporelle de modèles

La vérification de modèles a été définie par Clarke et Emerson [29] et de manière indépendante par Sifakis [102] au début des années 80. Elle vérifie si une description matérielle satisfait une propriété temporelle [43]. Une propriété décrit le comportement qu'un système doit avoir :

- toujours, finalement, jamais
- pour un chemin d'exécution ou pour tous

Deux types de comportements peuvent êtres spécifiés :

- combinatoire : référence à l'état courant du système, exprimé avec la logique propositionnelle
- séquentiel : référence au comportement dans le temps, exprimé avec la logique temporelle

La logique temporelle est la logique propositionnelle enrichie avec les opérateurs temporels : next, always, eventually, never. Il existe plusieurs variantes de logiques temporelles : *linéaires* comme LTL[2] [82] qui expriment des propriétés sur les états d'un chemin d'exécution, ou *arborescentes* comme CTL[3] [29] et CTL* [30] qui permettent d'exprimer des propriétés sur les arbres d'exécution du système.

Les propriétés sont classifiées en propriétés de sûreté et propriétés de vivacité [81]. Les propriétés de *sûreté* spécifient ce qui ne doit pas se produire (les mauvaises choses ne doivent

[1] Ordered Binary Decision Diagram
[2] Linear Temporal Logic
[3] Computation Tree Logic

jamais arriver) ou de manière équivalente, ce qui doit toujours se produire (toujours vrai dans tous les états). Les propriétés de *vivacité* spécifient ce qui devrait à un certain moment se produire (les bonnes choses arrivent finalement). Pour les systèmes finis, de nombreuses propriétés intéressantes en pratique peuvent être traduites vers des propriétés de sûreté [119]. Un contre-exemple pour les propriétés de sûreté est une trace d'états, ou le dernier état contredit la propriété. Pour les propriétés de vivacité, un simple contre-exemple est un chemin vers une boucle qui ne contient pas l'état demandé. Cette boucle représente un chemin infini qui n'atteint jamais l'état spécifié.

Les premiers algorithmes pour la vérification temporelle de modèles énumèrent explicitement les états atteignables du système pour vérifier la correction d'une spécification [31]. Une implémentation typique écrit les états individuels dans un grand tableau de hachage, en mémorisant les états atteints durant une recherche en profondeur de l'espace d'états. Comme le nombre d'états est généralement très grand (le nombre d'états d'un système est 2^n, où n est le nombre de variables booléennes du système), les premières implémentations ne sont capables de traiter que des descriptions de petite taille, et ne peuvent pas s'appliquer à des descriptions industrielles.

Une solution à ce problème est la *vérification symbolique de modèles* [35, 88] qui travaille avec un ensemble d'états à la place des états individuels et représente les ensembles d'états symboliquement, en utilisant des fonctions booléennes. La manipulation des formules booléennes, est réalisée d'une manière efficace en utilisant les OBDD ou plus court BDD [20], une représentation canonique en forme de graphe de fonctions booléennes. La combinaison de la vérification de modèles avec les BDD a permis de vérifier des systèmes avec 10^{20} états et plus [68].

Généralement, l'algorithme est le suivant : l'ensemble d'états initiaux est représenté par un BDD. A chaque pas i, l'ensemble des nouveaux états atteints est ajouté au BDD. A chaque nouveau pas, l'ensemble des nouveaux états est intersecté avec l'ensemble d'états qui satisfont la négation de la propriété. Si l'ensemble résultant est non vide, cela veut dire qu'une erreur a été détectée. Le processus finit quand l'ensemble des nouveaux états est vide ou une erreur a été trouvée. Dans le premier cas la propriété est vraie, puisque aucun état atteignable ne la contredit. Dans le deuxième cas un contre-exemple est renvoyé. La terminaison de l'algorithme est garantie, car le nombre d'états est fini. Il existe des algorithmes de recherche en avant ou en arrière en fonction de la propriété à prouver. Les algorithmes de recherche en avant commencent avec l'état initial et calculent les états futurs en appliquant la fonction de transition (comme décrit plus haut). Dans le cas des algorithmes de recherche en arrière, à partir des états qui contredisent la propriété, on montre qu'ils ne sont pas atteignables à partir de l'état initial. Dans la pratique, les algorithmes de recherche en avant sont plus rapides [53], mais certaines propriétés ne peuvent pas être résolues avec cet algorithme, d'où la nécessité de l'algorithme de recherche en arrière.

Un des inconvénients de cette méthode est la taille de la mémoire demandée pour le stockage et la manipulation des BDD. Les fonctions booléennes représentant l'ensemble d'états augmentent exponentiellement avec le nombre de variables booléennes nécessaires

pour le codage des objets du circuit. Même si diverses techniques, comme la décomposition [42], l'abstraction, et les réductions, ont été proposées pour résoudre ce problème, la vérification complète des nombreux systèmes est encore au-delà des capacités des vérificateurs symboliques basés sur les BDD.

La *vérification bornée de modèles* [13] basée sur les méthodes SAT [4, 28, 12, 11] est de plus en plus utilisée comme technique complémentaire à la vérification symbolique basée sur les BDD. Elle ne résoud pas le problème d'explosion d'états de vérificateurs de modèles, étant toujours une technique basée sur une procédure exponentielle, mais l'expérience a montré que dans certains cas elle peut résoudre des problèmes qui ne pourraient pas être abordés par les techniques basées sur des BDD. L'inverse est aussi valable, certains problèmes sont mieux résolus par des techniques basées sur les BDD. L'idée de la vérification bornée de modèles consiste, pour une machine à états finis et une propriété de la logique temporelle à vérifier, à chercher un contre-exemple pour la propriété dans l'espace de toutes les exécutions de la machine d'une longueur plus petite qu'un entier k. Si aucune erreur n'est trouvée, k peut être incrémenté jusqu'à ce qu'une erreur soit trouvée, que le problème explose, ou qu'une borne supérieure pour les chemins soit atteinte. Ce problème peut être réduit à un problème de satisfiabilité logique et être résolu par des méthodes SAT qui n'ont pas de problèmes d'explosion d'espace d'états comme les BDD. La limite k de l'algorithme est fixée par l'utilisateur. La méthode est incomplète si le nombre n'est pas suffisamment grand, puisque l'absence d'erreur n'est pas prouvée pour le comportement complet du circuit. Les expérimentations ont montré qui si k est suffisamment petit (moins de 80 cycles, en fonction du modèle et du solveur SAT), la vérification bornée de modèles donne de meilleurs résultats que la vérification symbolique de modèles avec BDD.

1.2.3 Avantages et inconvénients

Un des plus gros avantages de la vérification de modèles est son automatisation : la preuve d'une propriété ne dépend pas des interactions avec des utilisateurs et, si la propriété n'est pas vraie, un contre-exemple est généré automatiquement sous forme de trace. La vérification de modèles s'est avérée très efficace quand elle est appliquée aux circuits numériques séquentiels et aux protocoles de communication. Mais cette technique est applicable seulement pour les systèmes finis. Des travaux récents ont essayé d'appliquer l'algorithme de vérification de modèles pour des systèmes infinis comme les systèmes temps-réel [5] ou du logiciel [6], en utilisant des techniques d'abstraction.

Un autre avantage est la capacité de décrire des propriétés dynamiques grâce à la logique temporelle, combinée avec une méthode de raisonnement adaptée aux opérateurs temporels. En même temps, l'utilisation d'une logique temporelle est aussi un désavantage en raison du fait qu'elle n'est pas adaptable à la vérification de propriétés algorithmiques.

Un des plus gros inconvénients de la technique reste le problème d'explosion d'états. Avec la complexité croissante de circuits, le niveau d'abstraction pour la description monte. En conséquence, le modèle booléen sur lequel s'appuie la technique devient insuffisant. Donc le challenge technique dans la vérification de modèles est toujours de concevoir des

algorithmes et des structures de données qui permettent de réaliser des recherches sur de très grands espaces d'états.

Les premiers vérificateurs de modèles ont été conçus dans le monde académique, les plus connus étant SMV, BMC et VIS. La vérification temporelle de modèles est la deuxième méthode, après la vérification d'équivalence, à rentrer dans le monde industriel. De nombreux outils qui prennent en entrée du Verilog ou du VHDL existent sur le marché de la CAO de circuits : RuleBase d'IBM, FormalCheck de Cadence, Magellan de Synopsys.

Pour plus de détails sur la vérification de modèles voir [13, 32, 38].

1.3 La vérification basée sur la démonstration de théorèmes

Dans la vérification basée sur la démonstration de théorèmes, la spécification et l'implémentation sont représentées par des axiomes, et la relation qui existe entre elles est un théorème qui doit être prouvé dans la logique choisie. Les démonstrateurs de théorèmes n'ont pas été conçus spécialement pour répondre à la problématique de vérification de systèmes numériques. Ce sont des outils beaucoup plus généraux, basés sur un ensemble d'axiomes et sur un ensemble de règles de déduction logique.

Il existe plusieurs démonstrateurs de théorèmes et beaucoup d'entre eux ont été utilisés avec succès pour la vérification du matériel : ACL2 [72, 75, 19], PVS[99, 100, 55], HOL, Isabelle, NQTHM, Coq[9, 36] etc. Ils se différencient par le style de preuve, par la logique utilisée, par la manière dont les procédures de décision sont intégrées dans le système et par l'interface utilisateur.

Deux styles de preuve se distinguent :

- à partir des axiomes et hypothèses, les règles de déduction sont appliquées afin d'obtenir le théorème à prouver ;
- à partir du théorème à prouver, les inverses des règles de déduction sont appliqués afin d'obtenir des buts plus simples. Ceux-ci sont résolus avec des procédures de décision, ou sont réduits vers des axiomes ou des hypothèses.

Les types de vérification réalisés dans les méthodes basées sur la démonstration de théorèmes sont très variés. Le plus souvent sont vérifiées des propriétés de conformité, mais des propriétés d'équivalence et des propriétés logiques sont aussi facilement prouvables. Les propriétés temporelles ne font pas partie des propriétés traditionnelles, mais il existe des travaux qui les intègrent dans la vérification basée sur la démonstration de théorèmes ([85]).

Les logiques utilisées par les outils sont la logique du premier ordre et la logique d'ordre supérieur. Dans les deux types de logiques il est possible d'utiliser des variables booléennes et entières, et des fonctions. La quantification sur les variables est aussi possible. La quantification sur les fonctions n'est possible que dans la logique d'ordre supérieur. L'utilisation d'une logique ou de l'autre influence la manière dont les spécifications ou les descriptions matérielles sont modélisées, car les procédures de décision et les stratégies de preuve sont différentes. Le choix d'une logique n'est pas évident, les deux ont été appliquées avec succès aux problèmes de vérification matérielle. La logique du premier ordre est moins expressive,

mais a l'avantage d'être plus facile à automatiser. Aussi, la plupart des problèmes de vérification de circuits se réduisent à des problèmes exprimables avec la logique du premier ordre.

Dans la suite de cette section nous résumons les modèles de descriptions matérielles les plus utilisés dans la vérification basée sur la démonstration de théorèmes. Généralement, un circuit est modélisé par une fonction qui a comme arguments les entrées du circuit et retourne les sorties. Dans la logique d'ordre supérieur le circuit peut être modélisé aussi comme un prédicat. Comme les démonstrateurs de théorèmes du premier ordre sont basés sur une logique exécutable, il est plus intéressant dans ce cas de modéliser le circuit comme une fonction, afin de réaliser des simulations.

Si le comportement du circuit n'est pas dépendant du temps, le modèle est une fonction non récurrente :

$$System : \mathcal{T}_{input} \rightarrow \mathcal{T}_{output}$$

$$System(input) = output$$

Dans la logique d'ordre supérieur, le système peut être modélisée comme un prédicat :

$$System : \mathcal{T}_{input} \times \mathcal{T}_{output} \rightarrow \mathcal{B}$$

$$System(input, output) = (output = f(input))$$

Si le comportement du circuit dépend du temps, généralement des éléments de mémoire apparaissent. Il existe deux modélisations possibles.

- Le système est une collection de fonctions mutuellement récurrentes. Les entrées, les sorties et les éléments de mémoire sont définis comme des fonctions du temps.

  $$object : \mathbb{N} \rightarrow \mathcal{T}_{object}$$

  $$object(t) = f(in_i(t-1), \ldots, s_i(t-1))$$

  object est soit une sortie, soit un élément mémorisant, et in_i, s_i sont des entrées, respectivement des éléments mémorisants. Ce style de modélisation est plus adapté pour les démonstrateurs d'ordre supérieur, car la quantification sur des fonctions est possible.
- Le circuit est modélisé comme une machine à états.

  $$System : \mathcal{T}_{input} \times \mathcal{T}_s \rightarrow \mathcal{T}_{output}$$

  Step est une fonction qui calcule l'état suivant (la fonction de transition).

  $$Step : \mathcal{T}_s \times \mathcal{T}_e \rightarrow \mathcal{T}_s$$

  Dans la logique d'ordre supérieur le système est défini par :

  $$System(s, e) = \forall t \in \mathbb{N}\ s(t+1) = Step(s(t), e(t))$$

Dans la logique du premier ordre le système est une fonction récurrente :

$$System(s,e,n) = \begin{cases} s, \text{ si } n \leq 0 \\ System(Step(s,e),e,n-1), \text{ autrement} \end{cases}$$

Pour obtenir ces modèles à partir d'une description matérielle, plusieurs approches existent dans la littérature.

Une première approche est la modélisation manuelle du circuit dans la logique du démonstrateur, en tenant compte de sa structure et de son comportement [67, 114, 107]. Cette approche n'est pas acceptable par les concepteurs, car elle implique de refaire un modèle dédié pour le démonstrateur, différent de celui qui est l'entrée classique des outils de conception. De plus, cette approche peut induire des "faux positifs". Etant donné qu'il n'existe pas de lien direct entre le circuit réel et sa représentation logique, la preuve que le modèle est correct n'implique pas que le circuit est aussi correct. Pour répondre à ces problèmes, d'autres méthodes ont été proposées. Celles-ci sont basées sur la description des circuits directement dans la logique du démonstrateur et sur la traduction de ce modèle vers un langage de description matérielle, utilisé dans le flot classique de conception. Le circuit obtenu est correct par construction et peut être directement intégré dans le flot de conception ou utilisé comme un modèle de référence pour un circuit conçu par le flot classique [66, 141].

La deuxième approche est d'enfouir sémantiquement le langage de description matérielle utilisé dans le processus de conception dans la logique du démonstrateur. Ceci offre plusieurs avantages [18] : une définition formelle de la sémantique du langage ; le support pour la syntaxe et la vérification de types ; un environnement pour établir des méta-théorèmes sur le langage (par exemple, la cohérence) ; le support pour la preuve formelle des descriptions ; la dérivation des règles de preuve pour le langage ; la vérification des compilateurs pour le langage.

Il existe deux manières d'enfouir un langage de description matérielle dans la logique. La première, appelée enfouissement profond (*deep embedding*), consiste à représenter la syntaxe abstraite du langage par des termes logiques et à définir ensuite, dans la logique, les fonctions sémantiques qui donnent une signification aux descriptions. La deuxième modalité, appelée enfouissement superficiel (*shallow embedding*), consiste à définir seulement les opérateurs sémantiques dans la logique et créer une interface qui traduit directement la syntaxe du langage dans les structures sémantiques, et à l'inverse, les représentations sémantiques dans la syntaxe du langage.

Chaque type d'enfouissement a des avantages et des inconvénients. L'avantage de l'enfouissement profond est le fait qu'il permet de raisonner sur le langage, car la quantification sur les structures syntaxiques est possible dans la logique d'ordre supérieur. Mais la définition des termes pour la syntaxe abstraite et la définition des fonctions sémantiques sont très fastidieuses. L'avantage de l'enfouissement superficiel est que tout ce travail est évité, car l'interface gère la correspondance entre la description et sa représentation sémantique. Aussi, par rapport à l'approche précédente, les notations complexes sont plus facilement

traitables, car certains détails du langage sont éliminés dans le processus de traduction. En même temps, comme la correspondance est réalisée en dehors de la logique, elle n'est pas le sujet d'une spécification formelle et de preuve, donc elle est moins sûre. Dans ce cas, seules des propriétés sur la sémantique de description peuvent être prouvées, sans avoir la possibilité de raisonner sur des structures syntaxiques.

De manière générale, la partie la plus importante dans cette approche est la définition d'une sémantique. Il existe des langages de description matérielle avec une sémantique définie formellement, mais ce sont des langages académiques, qui ne constituent pas des entrées pour les outils de conception, et qui ne sont pas utilisés dans la pratique. Pour les langages industriels comme VHDL, Verilog, SystemC, il existe des travaux qui définissent des sous-ensembles de ces langages pour lesquels une sémantique formelle est fournie. Quelques-uns concernent la définition d'une sémantique pour VHDL pour l'application de la vérification basée sur la démonstration de théorèmes :

- Dans [130], Tassel a proposé une sémantique opérationnelle du cycle de simulation d'un sous-ensemble du VHDL avec un système de temporisation simplifié. Il réalise un enfouissement profond dans le démonstrateur HOL. Son modèle prend en compte le délai delta mais ne prend pas en compte les variables et les signaux résolus. Seule la preuve sur des circuits simples est envisageable, à cause de la complexité des détails induits par l'enfouissement profond.
- Dans [113], Russinoff définit une sémantique d'un sous-ensemble de VHDL comprenant les éléments de description structurelle et de flot de données, mais n'incluant pas la partie programmation séquentielle. Le temps est représenté par un couple d'entiers naturels dont la première composante représente le temps physique et la seconde le temps delta. Un signal est représenté par une liste d'évènements, un évènement étant un couple (v,t) signifiant qu'un signal prend la valeur v au temps t de simulation. La sémantique a été enfouie superficiellement dans le démonstrateur Nqthm et a permis la preuve de propriétés comportementales sur des simulations possibles des circuits combinatoires et séquentiels simples. Le sous-ensemble VHDL traité reste limité : seule l'affectation concurrente de signal et la gestion des composants sont considérées.
- Dans [47], Georgelin propose une formalisation de l'algorithme de simulation de VHDL à l'aide du démonstrateur ACL2 pour un sous-ensemble du VHDL synthétisable. Le circuit est modélisé comme une machine à états finis : l'état est défini comme une liste ACL2 qui contient les valeurs des variables et signaux du circuit ; la fonction de transition est une traduction directe de la description VHDL en Lisp et exprime le comportement du circuit pour un cycle d'horloge. Donc l'approche est applicable seulement aux circuits dont la partie combinatoire est ordonnée statiquement par compilation. L'algorithme de simulation est défini comme une fonction récursive qui peut être exécutée dans ACL2 numériquement ou symboliquement. La performance de cette approche est très limitée, la taille de la fonction de transition manipulée symboliquement étant très grande.

La sémantique de VHDL est donnée en termes de simulation, donc ceci est incontournable si nous voulons raisonner sur une description matérielle décrite en VHDL. Dans les approches précédentes, les auteurs ont enfoui la sémantique du langage dans un démonstrateur afin de profiter de ses heuristiques pour la réécriture, et réaliser la simulation symbolique et la preuve. Nous proposons une séparation des deux aspects : nous réalisons la simulation symbolique en utilisant un outil externe, et le résultat est traduit par la suite dans le démonstrateur de théorèmes. Dans ce cas, nous ne sommes plus dépendants d'un démonstrateur, l'utilisateur étant libre de choisir l'outil approprié, en fonction de sa problématique de vérification. En même temps, la simulation est réalisée par un outil dédié, plus rapide et efficace.

1.4 La simulation symbolique et les procédures de décision

La simulation symbolique [26], est une extension de la simulation classique et consiste à simuler le circuit pendant un nombre fini de cycles, en utilisant des symboles pour les entrées et pour les valeurs initiales des éléments mémorisants. Ainsi, une seule simulation correspond à plusieurs simulations numériques. En utilisant des techniques de vérification formelle, les expressions obtenues par la simulation symbolique de l'implémentation sont comparées avec des expressions soit obtenues par simulation symbolique de la spécification, soit fournies par le concepteur. La technique n'a pas eu beaucoup de succès au début car les inconvénients d'une simple exécution symbolique étaient trop forts : les expressions augmentent de manière exponentielle avec le nombre de cycles de simulation et deviennent illisibles. Si le programme contient des branchements conditionnels, l'arbre de simulation doit en tenir compte et sa taille augmente aussi de manière exponentielle.

Pour trouver des solutions à ces problèmes, trois façons de réaliser la simulation symbolique se sont distinguées : les expressions sont codées par des BDD, les expressions sont des formules dans la logique avec des fonctions non interprétées, les expressions sont des formules dans la logique classique avec des fonctions interprétées.

La première catégorie s'inspire de la vérification de modèles (section 1.2). Le circuit est modélisé comme une machine à états finis, et les fonctions de transition sont codées avec des BDD. Même si les BDD sont des structures assez compactes, durant la simulation symbolique leur taille grandit exponentiellement, d'où la nécessité d'heuristiques pour les réduire. Une façon de le faire est de remplacer certaines variables booléennes par des valeurs numériques, spécialement celles qui font partie du contrôle. Une autre solution est d'utiliser des expressions symboliques ternaires [22] ce qui implique des BDD réduites. L'avantage de cette approche est son degré d'automatisation, mais elle reste efficace seulement pour des circuits décrits au niveau booléen. Pour les circuits décrits à un niveau plus haut, le codage des expressions devient trop important.

Dans ce cas, les détails des unités fonctionnelles peuvent être abstraits en utilisant la logique avec des fonctions non interprétées (LEUF[4] [62]). Dans cette logique l'égalité entre

[4]Logic of Equality with Uninterpreted Functions

deux formules est simplement une égalité syntaxique, les fonctions n'étant pas évaluées. Par exemple, la valeur de vérité de la formule $a + a = 2 * a$ est faux, car les fonctions + et * ne sont pas évaluées.

Pour modéliser un circuit dans la logique LEUF, toute entrée du circuit est associée avec un terme si le type de l'entrée est scalaire ou avec une formule si le type de l'entrée est composé. Les éléments mémorisants sont affectés avec une expression symbolique extraite du comportement du circuit. Le système est vu comme un ensemble de traces à explorer et non comme une machine à états finis, donc il est moins sensible à l'espace des états que les techniques basées sur FSM.

Dans [132] est présenté un simulateur symbolique qui prend en entrée une description matérielle pipelinée et une spécification non-pipelinée, décrites dans un langage propre, AbsHDL. L'outil a besoin aussi d'un script de simulation qui contient des informations sur comment simuler l'implémentation et la spécification et quand comparer leurs états. Le résultat de la simulation est une formule dans la logique LEUF, qui exprime le critère de correction de l'implémentation par rapport à la spécification. La valeur de vérité de la formule est établie à l'aide de EVC[5], une procédure de décision dédiée à la logique LEUF [133].

UCLID est un autre outil basé sur la logique CLU[6] [21]. Ceci est une extension de LEUF avec des expressions lambda et une arithmétique dans laquelle toute somme a au moins un de ses arguments constant. L'outil combine la simulation symbolique avec une procédure de décision pour la logique CLU [80]. Pour les deux outils la spécification est une abstraction temporelle et/ou comportementale de l'implémentation.

Même si ces approches ont montré leur efficacité dans la vérification de systèmes complexes, leur inconvénient majeur est le manque d'expressivité des logiques employées, ce qui crée un gap sémantique entre la description intuitive d'un système et le modèle utilisé. Aussi, les concepteurs doivent abstraire leur circuit manuellement et l'exprimer dans le langage des outils. Ceci peut induire des erreurs dans la description et/ou cacher de vraies erreurs, dues aux modifications réalisées durant la modélisation.

Plusieurs démonstrateurs de théorèmes ont été utilisés comme des simulateurs symboliques. Les systèmes sont modélisés dans le démonstrateur et simulés symboliquement pour un nombre fini de cycles. Des techniques de réécriture et généralisation sont appliquées afin de réduire le plus possible la taille des expressions. Dans [108] est présenté l'enfouissement superficiel d'une sémantique fonctionnelle exécutable d'un sous-ensemble VHDL appelé VHDL- dans le démonstrateur Nqthm. Ce sous-ensemble comprend les variables, les signaux, l'instanciation de composants, l'affectation concurrente de signal, les processus et les sous-programmes. Les instructions wait ne sont pas prises en compte. Le moteur de simulation est exécutable, ce qui a permis la comparaison des résultats obtenus avec ceux des simulateurs commerciaux tels que celui de Cadence. En revanche, l'aspect preuve formelle n'est pas abordé à cause de la difficulté de raisonner sur ce simulateur formel. Dans [50], PVS est utilisé pour la simulation d'un microprocesseur JEM1 qui exécute du JAVA.

[5] Equality Validity Checker

[6] Counter arithmetic with Lambda expression and Uninterpreted functions

L'approche est généralisée pour ACL2 dans [92]. L'inconvénient dans ce cas est que le modèle est écrit manuellement et donc la performance de la simulation symbolique dépend de l'utilisateur et de ses connaissances sur le démonstrateur.

Dans sa thèse [115], Ghiath Al Sammane décrit un simulateur symbolique pour VHDL implémenté en Mathematica. Ce simulateur prend en entrée un modèle SRE[7] du VHDL (que nous définissons dans le chapitre 3) et un script de simulation. Le résultat de la simulation est un SRE que nous traduisons dans un démonstrateur de théorèmes.

Le modèle SRE est facilement traduisible dans les logiques avec des fonctions non interprétées mentionnées plus haut, en offrant ainsi un moyen de lier le langage industriel VHDL aux outils et techniques développés pour ces logiques. De l'autre coté, les logiques avec des fonctions non interprétées ne sont pas suffisamment expressives. Comme présenté auparavant, les opérations arithmétiques sont traitées comme des fonctions non interprétées, donc de nombreuses vérifications ne sont pas possibles. En conséquence, nous considérons que la portée des types de vérification de ces outils est assez limitée, même s'il sont automatiques, d'où notre choix d'utiliser des démonstrateurs de théorèmes.

[7] System of Recurrence Equations

Chapitre 2

Modélisation d'un circuit par un système d'équations récurrentes

2.1 Les langages de spécification de matériel

Les langages de description matérielle (HDL) sont utilisés pour décrire différents aspects des composants numériques. Le premier formalisme proposé par Reed [109] dans les années 50, est une liste de fonctions booléennes qui définissent l'entrée pour un bloc de bascules synchronisées par une horloge. A partir des années 70, de nombreux langages sont définis [16, 27]. Dans les années 80 apparaissent VHDL et Verilog. Verilog est apparu comme un langage propriétaire d'un produit commercial, tandis que VHDL a été défini par le Département de la Défense des Etats Unis [39]. Les deux deviennent des standards IEEE : VHDL en 1987 [56], révisé en 1993 [57] et 2000 [59] et Verilog en 1995 [58], révisé en 2001 [60]. Il y a toujours eu une concurrence entre ces deux standards, pourtant utilisés d'une manière équilibrée dans le monde industriel et restant les langages de référence jusqu'à maintenant.

SystemC, initialement défini par Synopsys [83], est utilisé comme langage de spécification de systèmes depuis ces dernières années. SystemC a été initialement conçu pour remplacer le niveau RTL, afin de ne plus avoir besoin d'un autre langage de description (VHDL ou Verilog) et d'accéder directement à la synthèse. Mais son but a été détourné, il est maintenant utilisé principalement pour modéliser les systèmes électroniques au niveau système, avant le partitionnement matériel/logiciel.

Un langage de description matérielle contient des caractéristiques propres aux langages de programmation comme les types de données et les instructions séquentielles, mais aussi des constructions propres à la conception numérique : la capacité de décrire la structure d'un système comme l'interconnexion de plusieurs composants, chaque composant ayant une interface et plusieurs implémentations possibles ; la notion de parallélisme : des parties du système sont exécutées en parallèle et elles peuvent communiquer afin de se synchroniser ; la notion de temps.

Ces concepts se retrouvent dans les trois langages industriels Verilog, VHDL, SystemC. Même si leurs syntaxes sont très différentes, leur usage par les concepteurs a convergé vers

un point commun, avec la même sémantique. Aujourd'hui il existe plusieurs outils qui font la traduction entre les trois langages.

Par la suite nous allons utiliser la syntaxe de VHDL, sachant que le traitement peut s'adapter facilement au Verilog ou SystemC.

2.2 Le sous-ensemble VHDL

A cause de la complexité sémantique de VHDL, nous sommes restreints à un sous-ensemble de ce langage.

Une première restriction est le temps. Nous ne prenons pas en compte le type TIME de VHDL. Donc les affectations de signal acceptées sont seulement à délai nul. Aussi la forme de l'instruction wait est plus restreinte, n'acceptant pas d'attente sur un nombre d'unités de temps. Le temps physique est ignoré pour deux raisons principales : d'une part, dans la pratique, la grande majorité des descriptions matérielles ne l'utilisent pas, car la synthèse ne le prend pas en compte, d'autre part sa considération aurait beaucoup compliqué le modèle et la manière d'appliquer des techniques de simulation symbolique ou de vérification.

Une autre restriction est le non déterminisme. Dans le standard IEEE VHDL'93 sont introduites des variables partagées qui peuvent être déclarées à l'extérieur d'un processus. Cette notion peut engendrer du non déterminisme, donc nous ne le traitons pas.

Nous traitons seulement les descriptions qui se stabilisent. D'après le modèle de simulation, le programme à l'intérieur d'un processus s'exécute infiniment. Si à l'intérieur du processus il n'existe pas une instruction de synchronisation (wait), le simulateur ne sort jamais de ce processus, et la simulation est bloquée : les autres processus ne sont plus exécutés et le temps de simulation n'avance plus. Afin d'éviter cette configuration, nous supposons que tous les processus ont au moins une instruction wait. Nous rappelons qu'un processus avec une liste de sensibilité se réécrit dans un processus avec une instruction wait.

Le sous-ensemble VHDL que nous prenons en compte est un peu plus large que le sous-ensemble synthétisable de VHDL[61], incluant des structures de données non bornées, des boucles *while* et des boucles *for* non bornées, des fonctions récursives, et n'ayant pas de restrictions ayant pour but une reconnaissance syntaxique de l'horloge.

VHDL est un langage basé sur un modèle entité - architecture.

L'entité décrit la vue externe du composant. Elle associe un nom au composant et une interface. L'interface énumère les ports d'entrée, de sortie ou d'entrée-sortie avec leurs types. Une entité peut avoir aussi des déclarations de constantes, de types ou des paramètres génériques permettant de décrire des classes de composants. A une entité peuvent être associées plusieurs architectures.

L'architecture décrit le comportement du composant. Elle contient une partie déclaration et une partie instruction. Dans la première partie peuvent être déclarés des types, constantes, fonctions, procédures, composants et signaux. La deuxième partie contient des instructions concurrentes qui sont exécutées en parallèle, donc l'ordre dans lequel elles sont

écrites n'a pas d'importance. Il y a plusieurs styles de description de l'implémentation d'un composant. Dans le style comportemental, l'architecture est décrite d'une manière algorithmique comme une collection de processus. Dans le style structural, l'architecture est un ensemble de composants interconnectés. Dans le style flot de données, l'architecture est une collection d'affectations concurrentes de signaux. Généralement, dans une description, les styles sont combinés.

Une configuration spécifie le couple entité - architecture afin d'instancier un composant.

Le paquetage rassemble des déclarations et des définitions de types, constantes, sous-programmes, composants afin de permettre leur utilisation par plusieurs descriptions VHDL. Il existe des paquetages standard comme STD_LOGIC_1164 qui définit un type logique avec neuf valeurs et les opérateurs associés, ou le paquetage STANDARD du langage VHDL.

Les types de base de VHDL sont entier, booléen, réel et le type énuméré. Parmi les types composés sont acceptés des tableaux contraints indexés par des types énumérés ou entiers bornés, des tableaux non bornés et des structures. Les types fichiers n'ont pas de signification pour la vérification, donc nous les ignorons. Les pointeurs peuvent être considérés comme des tableaux non bornés. Des travaux ont été menés dans le cadre de la synthèse de circuits à partir de C, pour la correspondance d'un pointeur avec un indice de tableau ([120]).

Les expressions en VHDL sont construites avec des opérateurs arithmétiques, relationnels, logiques, de concaténation et des appels de fonctions. Une fonction est appelée à l'intérieur d'une expression et retourne une seule valeur au même temps de simulation qu'elle a été appelée.

Les objets de base du langage sont les constantes, les signaux et les variables. Chaque objet est déclaré avec un type, et éventuellement une valeur initiale. Une variable a le même comportement que dans un langage de programmation : toute affectation a comme résultat le changement immédiat de sa valeur. Les variables sont déclarées seulement à l'intérieur d'un processus ou d'un sous-programme. La sémantique du signal est étroitement liée à la notion de temps en VHDL : le résultat d'une affectation n'est pas le changement immédiat de sa valeur, mais un changement dans son "pilote" qui sera effectif dans un instant futur. Les signaux ne sont pas déclarés à l'intérieur de processus ou sous-programmes.

Les instructions séquentielles sont de trois types :

- les instructions de contrôle qui sont des instructions conditionnelles *if then else*, *case*, ou de boucle *while*, *for* ;
- les instructions d'affectation sont de deux types : l'affectation de variable et l'affectation de signal. Pour la deuxième nous considérons seulement l'affectation avec un délai nul ;
- l'instruction de synchronisation wait ;
- l'appel de procédure. Les procédures peuvent contenir des instructions de synchronisation.

Les instructions concurrentes considérées sont l'affectation concurrente de signal, l'affectation sélective de signal, l'affectation conditionnelle de signal, le processus et l'instanciation de composant. Un processus est une succession d'instructions séquentielles. Il contient soit une liste de sensibilité soit des instructions de synchronisation wait.

Du fait de la complexité syntactique de VHDL, nous préférons réécrire certaines instructions vers des formes plus primitives. Par exemple l'instruction case se réécrit facilement dans des *if then else* imbriqués. De même, les affectations de signal conditionnelles et sélectives sont réécrites en des affectations de signal avec des *if then else* imbriquées dans la partie droite.

Deux types de sous-programmes existent dans VHDL : les procédures et les fonctions. Les fonctions peuvent être pures ou impures. Nous traitons seulement la première catégorie, les fonctions étant éventuellement récursives. Les fonctions de résolution, un cas particulier de fonctions, sont utilisées pour définir la valeur d'un signal quand il est affecté par deux processus concurrents. Les procédures peuvent contenir des instructions de synchronisation.

La syntaxe du sous-ensemble VHDL que nous prenons en compte est présentée dans l'annexe 5. Le tableau 2.1 résume la syntaxe abstraite de ce sous-ensemble.

Pour VHDL nous considérons les catégories syntaxiques suivantes :
- les objets $\mathcal{O}$ avec les sous-catégories : les variables $\mathcal{V}$, les signaux $\mathcal{S}$ et les constants symboliques $\mathcal{C}$. Nous considérons Id le domaine des noms des objets.
- les expressions $\mathcal{E}xpr$ avec les sous-catégories : les expression arithmétiques E_{arith} et les expressions booléennes E_{bool} ;
- les instructions $Instr$ avec les sous-catégories : les instructions séquentielles Seq et les instructions concurrentes $Conc$;
- les déclarations $Decl$.

La simulation de VHDL

Dans le manuel de référence, la sémantique de VHDL est définie en termes de simulation.

Le premier pas dans l'exécution est l'élaboration. Durant cette étape les composants sont liés aux entités, la hiérarchie du modèle est mise à plat, et le résultat est des processus connectés par des signaux. Cette structure sera simulée en utilisant un algorithme de simulation basé sur des événements.

Le processus de simulation a deux étapes : une étape d'initialisation qui est réalisée après l'élaboration, et le cycle de simulation qui est répété jusqu'à la fin de la simulation.

Dans la phase d'initialisation, le temps courant de simulation est initialisé à 0, et chaque signal prend sa valeur initiale (donnée explicitement dans la déclaration, ou sinon sa valeur par défaut). Dans le cycle de simulation a lieu la mise à jour des signaux suivie de l'exécution de processus.

Le simulateur est responsable de la mise à jour de valeurs des signaux. Si la valeur ancienne est différente de la valeur courante, un événement correspondant au signal est

VHDL : := (Ent ; Arch)
Ent : := *name* (in, out, inout, Decl)
Arch : := *name* (Decl, Conc)

Decl : := **type** *name* **is** type_def
| **signal** *name* : type :=val
| **variable** *name* : type :=val
| **constant** *name* : type :=val
| type **function** *name* (params) Decl ;Seq **return** *expression*
| **procedure** *name* (params) Decl ;Seq

Config : := *name* (Ent ;Arch)

Conc : := *id* <= *expression*
| *component_name* (args)
| **process** [(sensitivity_list)] Decl ; Seq **end process**
| $Conc_1$; $Conc_2$

Seq : := *id* <= *expression*
| *id* := *expression*
| *proc_name* [(args)]
| **if** *condition* **then** Seq_1 [**else** Seq_2] **end if**
| **while** *condition* **loop** Seq **end loop**
| **for** *i* **from** a_1 dir a_2 **loop** Seq **end loop**
| **wait on** sensitivity_list **until** *condition*
| Seq_1 ; Seq_2

expression := a | b | **if** b **then** $expression_1$ [**else** $expression_2$] **endif**

$a := n \mid id \mid func\_name(e_1,...,e_n) \mid (a) \mid a_1\ \text{op}_{arith}\ a_2 \mid \text{abs}\ a \mid \text{-}\ a$
$\text{op}_{arith} := + \mid \text{-} \mid * \mid \& \mid / \mid \text{rem} \mid \text{mod} \mid **$

$b := \text{true} \mid \text{false} \mid id \mid func\_name(e_1,...,e_n) \mid (b) \mid a_1\ \text{op}_{rel}\ a_2 \mid \text{not}\ b \mid b_1\ \text{op}_{bool}\ b_2$
$\text{op}_{rel} := = \mid \neq \mid < \mid \leq \mid > \mid \geq$
$\text{op}_{bool} :=$ and | or | xor | nand | nor | xnor

$id := x \mid id_1.id_2 \mid id(a_1,...,a_n) \mid id(a_1\ \text{dir}\ a_2)$

dir := **to** | **downto**

TAB. 2.1 – Syntaxe abstraite des spécifications VHDL

généré. Si un processus contient une liste de sensibilité, toute modification des signaux de cette liste entraîne l'activation du processus. S'il contient des instructions wait, les instructions du processus sont exécutées jusqu'à ce qu'une instruction *wait* soit rencontrée. La rencontre d'une instruction *wait* suspend le processus. Il est à nouveau actif quand la condition de réveil est remplie. Quand tous les processus sont suspendus, le simulateur met à jour les valeurs de signaux, ce qui peut encore entraîner des événements. Un cycle de simulation qui a lieu au même temps de simulation que le précédent est appelé un cycle delta. Si la mise à jour des signaux n'entraîne plus d'événements, l'horloge universelle, qui a la valeur du temps de simulation courant, est incrémentée.

2.3 Des sémantiques formelles pour VHDL

De nombreuses sémantiques pour VHDL ont été définies afin de créer un cadre formel pour la vérification de descriptions [77]. Ceci est dû à la complexité et à l'ambiguïté du manuel de référence pour VHDL.

Les sémantiques proposées varient en fonction du domaine d'application, de la partie du langage traité et du modèle sémantique utilisé.

Dans [49] une sémantique opérationnelle est définie pour le standard IEEE VHDL'87, ce travail étant étendu au VHDL'93 par [123]. Dans [17] une sémantique fonctionnelle est proposée pour un sous-ensemble synchrone du VHDL nommé P-VHDL. La sémantique ne considère pas le temps physique, mais seulement le délai delta. L'apport de ces travaux est de bien séparer la phase d'élaboration de celle de simulation.

D'autres sémantiques pour VHDL ont été définies, ayant comme but l'interfaçage avec des méthodes de vérification formelles.

Pour la vérification de modèles :

- Dans [98] une sémantique complète est définie en termes de réseaux Petri colorés. Les réseaux générés sont d'une taille très importante et ne peuvent pas servir de support à des outils de vérification sans une simplification préalable.
- Une autre sémantique avec des réseaux Petri est définie dans [44] pour produire des modèles pour la preuve d'équivalence et de propriétés temporelles.
- Dans [37], Déharbe propose un sous-ensemble VHDL adapté pour la vérification de modèles et il définit sa sémantique en termes de machines d'état finis. La sémantique est utilisée pour vérifier des propriétés CTL.

Des sémantiques ont été également proposées afin de rendre possible l'enfouissement du VHDL dans des logiques de démonstrateurs de théorèmes :

- Van Tassel, [129], a proposé le premier une sémantique opérationnelle du cycle de simulation d'un sous-ensemble de VHDL pour HOL. Sa sémantique modélise le délai delta mais il ne prend pas en compte les variables et les signaux résolus.
- Dans [110], Reetz et al., considèrent un sous-ensemble constitué des éléments de base du langage et définissent une sémantique basée sur les graphes de flot d'exécution séparant données et contrôle. La sémantique est intégrée dans HOL et prend en compte le délai delta.

- Dans [108] une sémantique fonctionnelle exécutable d'un sous-ensemble VHDL est définie dans le démonstrateur Nqthm. L'aspect preuve formelle n'est pas abordé à cause de la difficulté de raisonner sur ce simulateur formel.
- Russinoff [113] définit une sémantique d'un sous-ensemble comprenant seulement l'affectation concurrente de signal et la gestion des composants. La sémantique a été implémentée dans le démonstrateur Nqthm.
- Georgelin [47] propose une formalisation de l'algorithme de simulation de VHDL à l'aide d'ACL2. Le circuit est modélisé comme une machine d'états finis où la fonction de transition est la traduction directe de la description VHDL en Lisp. L'algorithme de simulation est défini comme une fonction récursive qui peut être exécutée dans ACL2 numériquement ou symboliquement.

Dans certaines des approches mentionnées plus haut, les auteurs définissent la sémantique du VHDL directement dans la logique de l'outil de leur choix (par un enfouissement profond). A cause du grand nombre de détails sémantiques, il est très difficile de raisonner sur le résultat. D'autres auteurs ont choisi un enfouissement superficiel de la sémantique, c'est à dire de transformer les éléments syntaxiques de VHDL dans des concepts existant dans la logique (voir section 1.3) et d'appliquer ensuite des règles sémantiques correspondant à l'algorithme de simulation VHDL. Dans ce cas, il n'a jamais été démontré que les règles de traduction gardent la sémantique du VHDL. De plus, la difficulté de trouver la bonne transformation a fait que les sous-ensembles VHDL traités sont très restreints.

Pour cette raison, nous avons choisi de passer par des étapes intermédiaires. Tout d'abord, nous définissons un ensemble de règles de transformation afin d'obtenir un modèle compact et prouvable. Ensuite nous montrons comment simuler symboliquement le modèle conformément à l'algorithme de simulation de VHDL afin d'obtenir un modèle stable (sans cycles delta). Ce modèle est traduit par la suite dans la logique du démonstrateur.

Dans la section suivante, nous introduisons le formalisme d'équations récurrentes sur lequel est basé notre modèle pour les systèmes éléctroniques.

2.4 Présentation du modèle : système d'équations récurrentes

Observer le comportement d'un système matériel revient à observer ses sorties et ses éléments mémorisants durant le temps d'exécution. En conséquence, la valeur d'un objet x après t cycles de simulation est de la forme

$$x(t) = f(\overrightarrow{x}(t-1), \overrightarrow{x}(t-2), \ldots, \overrightarrow{x}(0)),$$

où $\overrightarrow{x} = (x_0, x_1, \ldots, x_m)$, et x_i sont des objets du système, $0 \leq i \leq m$. $\overrightarrow{x}(t')$ représente les valeurs des objets x_i au cycle t'.

La sémantique opérationnelle de VHDL est donnée en termes de cycles de simulation.

De ce point de vue, il est convenable de décrire le comportement d'un objet comme une équation récurrente sur le temps de simulation qui calcule la valeur de l'objet après un cycle de simulation. Conformément à la sémantique de simulation d'un signal, pendant le cycle de simulation t la valeur de l'objet x est calculée, qui sera valide pour le prochain cycle de simulation, $t+1$.

Ce calcul dépend éventuellement des événements générés durant la simulation. Un événement sur un signal est produit si sa valeur dans le cycle de simulation courant $x(t)$ est différente de sa valeur dans le cycle de simulation précédent $x(t-1)$. Donc pour calculer la valeur d'un objet x après un cycle de simulation, deux valeurs sont nécessaires : sa valeur courante (au temps t) et sa valeur précédente (au temps $t-1$) :

$$x(t+1) = f(\overrightarrow{x}(t), \overrightarrow{x}(t-1)), \forall t \in \mathbb{N}.$$

f est une fonction qui calcule la valeur de l'objet durant un cycle de simulation. Nous allons montrer dans ce chapitre comment extraire ces fonctions à partir d'une description VHDL.

Le formalisme d'équations récurrentes uniformes a été introduit par Karp dans [71] pour la spécification des calculs parallèles.

Définition 11 (Equation récurrente uniforme) *Une équation récurrente uniforme est*

$$t \in \mathbb{Z} \rightarrow x(t) = f(y(dep(t)), \ldots)$$

où :
- *t est un élément du domaine D de l'équation*
- *dep est la fonction de dépendance de la forme $dep(t) = t + d$, ou $d \in \mathbb{Z}$ est une constante.*
- *x et y sont des noms de variables*
- *f est une fonction*

Nous définissons un système d'équations récurrentes uniformes pour modéliser un circuit VHDL.

Définition 12 (Système d'équations récurrentes) *Un système d'équations récurrentes est un n-uplet :*

$$SER = \langle \mathcal{I}nput, \mathcal{L}ocals, \mathcal{O}ut, Init, \{E_x : x(t+1) = f_x(\overrightarrow{x}(t), \overrightarrow{x}(t-1))\}_{x \in \mathcal{L}ocals \cup \mathcal{O}ut}, \mathcal{D}ecl \rangle$$

- *$\mathcal{I}nput$ est l'ensemble des signaux d'entrée.*
- *$\mathcal{L}ocals$ est l'ensemble des signaux et variables internes.*
- *$\mathcal{O}ut$ est l'ensemble des signaux de sortie.*
- *$\mathcal{D}ecl$ est l'ensemble de déclarations du système : les déclarations de types, d'objets, de fonctions ; etc....*
- *$Init$ est la fonction d'initialisation des éléments d'état et de sorties. $x(0) = Init(x)$ et $x(-1) = x(0)$.*

- *E_x est l'équation récurrente de l'objet x. f_x est la fonction qui définit le comportement de l'objet pour une unité de temps.*
  $f_x : \mathcal{T}^n \rightarrow \mathcal{T}$, n est le nombre de paramètres de f_x.
- *$\overrightarrow{x} = (x_0, x_1, \ldots, x_m)$ sont des objets du système.*

Les types d'objets $\mathcal{T}$, sont les types définis en VHDL : entiers, réels, booléens, structures et tableaux.

Le premier modèle extrait à partir d'une description VHDL est au niveau de cycle de simulation, donc, dans ce cas, une unité de temps correspond à un cycle de simulation. L'unité de temps peut correspondre aussi à plusieurs cycles de simulation. Par simulation symbolique, nous calculons le comportement du circuit pour plusieurs cycles, ainsi nous pouvons considérer que cette exécution est effectuée dans une seule unité de temps. Ceci est une manière d'éliminer les cycles delta ou d'obtenir des systèmes d'équations récurrentes qui correspondent à un cycle d'horloge ou même à plusieurs cycles d'horloge.

2.5 Comparaison avec la machine d'états finis

L'espace de données est une différence importante entre notre modèle et la FSM. Le SER gère des domaines non bornés, tandis que la FSM modélise seulement des données avec des types finis. Les méthodes de vérification basée sur la FSM utilisent un modèle booléen de celui-ci, donc le nombre de variables de codage est très grand et le raisonnement appliqué est dans la logique propositionnelle. Dans le SER, les variables restent au niveau de la définition, et le raisonnement est porté dans la théorie à laquelle les données appartiennent.

Au niveau du temps, il existe aussi des différences. Le modèle du système d'équations récurrentes peut décrire des comportements plus détaillés que le modèle de machine d'états finis. Le modèle du système d'équations récurrentes modélise une description au niveau des événements, donc au niveau de delta cycles. Généralement, le calcul des valeurs suivantes pour les éléments mémorisants et pour les sorties dépend des valeurs courantes et des valeurs anciennes. Donc la fonction de transition pour le SER (qui est la réunion des fonctions des équations récurrentes) dépend de deux états. Dans le même temps, pour les circuits synchronisés par horloge, si le SER est simulé symboliquement pour un cycle d'horloge, la fonction de transition du SER résultant dépend seulement de l'état courant, car tous les événements sont éliminés.

De son côté, la machine d'états finis synchrone déterministe décrit le comportement du système seulement au niveau de cycle d'horloge. Nous avons trouvé plus intéressant un modèle au niveau de cycles de simulation, car il n'existe pas de restriction d'écriture comme c'est le cas pour la machine d'états, et nous pouvons donc traiter une classe plus large de descriptions matérielles.

Tout système d'équations récurrentes peut être codé comme une machine d'états finis s'il a les propriétés suivantes :

- toute équation récurrente est d'ordre 1 (f_x a comme argument seulement des fonctions de la forme $y(t)$, et non pas $y(t-1)$)

– l'espace de données est fini.

Le passage inverse de la machine d'état vers un SER est plus générale : toute machine d'états finis correspondant à un circuit est un SER.

Donc la FSM peut être vue comme un cas particulier de SER. Par exemple, si les entrées, les sorties et les états d'un système sont des variables booléennes, et le circuit est soit synchronisé par une horloge, soit purement combinatoire, le SER résultant après la simulation symbolique pour un cycle d'horloge, est équivalent avec la FSM booléenne correspondant au système.

2.6 Extraction d'un SER à partir d'une description VHDL

La méthode d'extraction des systèmes d'équations récurrentes prend comme entrée un programme en langage VHDL, et produit d'abord une liste d'affectations, exactement une pour chaque signal et variable affectée dans la conception. Dans un deuxième temps, la liste d'affectations est transformée dans un système d'équations récurrentes. La première étape est réalisée par la fonction **Trans**.

Le résultat de la fonction **Trans** est un ensemble d'affectations parallèles, **AP**, qui modélise comment les objets du système sont modifiés pendant un cycle de simulation.

AP est défini comme suit :

$$\mathbf{AP} := x := \mathrm{E} \mid \mathrm{AP}_1 \parallel \mathrm{AP}_2$$

– $x := \mathrm{E}$ est une affectation parallèle si x est une variable ou un signal et E est une expression ;
– la composition parallèle des deux affectations parallèles est une affectation parallèle.

Nous utilisons la notation $\|_{v \in V} S(v)$ pour décrire la composition parallèle des affectations $S(v)$. Soit $V = v_1, \ldots, v_n$, $\|_{v \in V} S(v) = S(v_1) \| S(v_2) \| \ldots \| S(v_n)$.

La sémantique opérationnelle d'**AP** est la suivante : l'affectation parallèle de signal modifie la valeur future présumée du signal, sans modifier les autres valeurs ; l'affectation parallèle de variable modifie la valeur courante de la variable. La composition parallèle de deux affectations modifie les valeurs courantes des variables et les valeurs futures des signaux affectés dans chaque affectation **AP**. Puisque, par définition, une affectation parallèle modifie un objet une seule fois, il n'existe pas de conflit.

Par la suite, nous présentons quelques notations et notions utilisées dans ce chapitre.

Notations

Nous notons par $\mathrm{W_P}$ la fonction qui renvoie l'ensemble des objets écrits par un programme P en VHDL.

$\mathrm{W} : \mathcal{S}eq \cup \mathcal{C}onc \cup \mathcal{I}d \rightarrow \mathcal{S} \cup \mathcal{V}$

$W_{id <= expression} = W_{id}$ $\quad W_{id := expression} = W_{id}$
$W_{\text{if } condition \text{ then } P_1} = W_{P_1}$ $\quad W_{\text{if } condition \text{ then } P_1 \text{ else } P_2} = W_{P_1} \cup W_{P_2}$
$W_{\text{while } condition \text{ do P enddo}} = W_P$ $\quad W_{\text{for } i \text{ from } a \text{ dir } b \text{ do P enddo}} = W_P \cup \{i\}$
$W_{P,Q} = W_P \cup W_Q$ $\quad W_{P \| Q} = W_P \cup W_Q$
$W_n = \emptyset$ $\quad W_x = \{x\}$ $\quad W_{id_1.id_2} = W_{id_1}$ $\quad W_{id(a_1,\ldots,a_n)} = W_{id}$
$W_{e \text{ op } f} = \emptyset$ $\quad W_{\text{op } e} = \emptyset$
$W_{name(\text{args})} = W_{name\_body}[\text{params/args}]$
$W_{\text{process [(sensitivity\_list)] begin Decl Seq endprocess}} = W_{\text{Seq}}$

L'objet écrit par une affectation est l'objet affecté. W appliquée à *if then else* retourne les objets écrits dans les deux branches et appliquée aux instructions de boucle retourne les objets écrits à l'intérieur de la boucle. Pour l'instruction *for* le paramètre de la boucle, i, en fait aussi partie. Les objets écrits par les procédures, les fonctions, les processus ou les composants, sont ceux écrits par leur corps. W appliquée à la composition séquentielle ou parallèle des blocs d'instructions retourne la réunion des objets écrits par chaque bloc. Pour les identificateurs, W retourne l'objet racine de l'élément auquel l'identificateur fait référence. Nous rappelons que nous considérons les objets de type composé (comme les tableaux, les enregistrements) comme une entité. Donc si une affectation d'un des éléments du tableau a lieu, nous considérons que l'objet écrit est le tableau lui-même et non pas explicitement l'élément. Pour les expressions, la fonction W retourne l'ensemble vide.

Nous étendons la définition de W sur les affectations paralléles.

$W : \mathcal{S}eq \cup \mathcal{C}onc \cup \mathcal{I}d \cup \mathbf{AP} \rightarrow \mathcal{S} \cup \mathcal{V}$

$W_{x := E} = \{x\}$ $\quad W_{S \| Q} = W_S \cup W_Q$

Dans le même temps nous allons noter par R_P l'ensemble d'objets lus par P, où P est un bloc d'instructions VHDL.

$R : \mathcal{S}eq \cup \mathcal{C}onc \cup \mathcal{E}xpr \rightarrow \mathcal{S} \cup \mathcal{V}$

$R_{id <= expression} = R_{expression}$ $\quad R_{id := expression} = R_{expression}$
$R_{\text{if } cond \text{ then } P_1} = R_{P_1} \cup R_{cond}$ $\quad R_{\text{if } cond \text{ then } P_1 \text{ else } P_2} = R_{P_1} \cup R_{P_2} \cup R_{cond}$
$R_{\text{while } cond \text{ do P enddo}} = R_P \cup R_{cond}$ $\quad R_{\text{for } i \text{ from } a \text{ dir } b \text{ do P enddo}} = R_P \cup \{i\} \cup R_a \cup R_b$
$R_{P,Q} = R_P \cup R_Q$ $\quad R_{P \| Q} = R_P \cup R_Q$
$R_n = \emptyset$ $\quad R_x = \{x\}$ $\quad R_{id_1.id_2} = R_{id_1}$ $\quad R_{id(a_1,\ldots,a_n)} = \cup_{i \in 1,n} R_{a_i} \cup R_{id}$
$R_{e \text{ op } f} = R_e \cup R_f$ $\quad R_{\text{op } e} = R_e$
$R_{name(\text{args})} = R_{name\_body}[\text{params/args}]$
$R_{\text{process [(sensitivity\_list)] begin Decl Seq endprocess}} = R_{\text{Seq}} \cup R_{sensitivity\_list}$

La fonction R appliquée aux affectations retourne les objets lus par l'expression affectée. Les objets lus par les instructions de contrôle *if then else* sont ceux lus par la condition et les blocs de deux branches. Pour les instructions de boucle, R retourne les objets lus

par la condition et ceux lus à l'intérieur de la boucle. Pour *for* le paramètre de la boucle, i, en fait aussi partie. Les objets lus par les procédures, les fonctions, les processus ou les composants, sont ceux lus par leur corps, après la réalisation de la substitution dans l'appel de fonction correspondant aux noms d'arguments. R appliquée à la composition séquentielle ou parallèle de blocs d'instructions retourne la réunion des objets lus par chaque bloc. Pour les identificateurs, R retourne l'objet racine de l'élément auquel l'identificateur fait référence. R appliquée à une constante numérique retourne l'ensemble vide. Pour les expressions, la définition de R suit leur structure.

Nous étendons la définition de R sur les affectations parallèles.

$\mathrm{R} : \mathcal{S}eq \cup \mathcal{C}onc \cup \mathcal{E}xpr \cup \mathbf{AP} \rightarrow \mathcal{S} \cup \mathcal{V}$

$\mathrm{R}_{x:=\mathrm{E}} = \mathrm{R}_E \qquad\qquad \mathrm{R}_{\mathrm{S}\|\mathrm{Q}} = \mathrm{R}_S \cup \mathrm{R}_Q$

Substitutions

Une substitution d'un objet x dans une expression E est notée par $E[x/v]$, où v est une expression nouvelle. Le résultat de la substitution est une nouvelle expression E', dans laquelle toutes les occurences de l'objet x sont remplacées par v. Les substitutions peuvent être séquentielles ou parallèles.

Une substitution séquentielle a la forme $E[x/v] \ldots [y/u]$ et son résultat est une expression dans laquelle d'abord toutes les occurences de x ont été remplacées par v, et ensuite les occurences de y ont été remplacées par u. Ainsi (a+b+x)[a/x][x/c] devient (c+b+c).

Une substitution parallèle a la forme $E[x/v, \ldots, y/u]$ et son résultat est une expression dans laquelle x et y sont simultanement remplacées par v, repectivement u. Ainsi, (a+b+x)[a/x, x/c] dévient (x+b+c).

Listes

Une liste est une structure définie recursivement :

- la liste vide, nil, est une liste ;
- soit a un élément, et l une liste, alors (a, l) est une liste.

Nous utilisons plusieurs opérations sur les listes :

- la concaténation, notée par &. L'opérateur & est associatif : $(A\&B)\&C = A\&(B\&C)$, mais il n'est pas commutatif $A\&B \neq B\&A$.
- l'appartenance d'un élément k à une liste l notée par $k \in l$.
- l'acces de l'élément i de la liste l, noté par $l(i)$.

Afin de faciliter la compréhension du mécanisme de transformation des circuits vers des affectations paralleles, nous avons choisi d'expliquer d'abord l'approche sur un sous-ensemble simple de VHDL. Ensuite, nous allons étendre ce sous-ensemble afin de prendre en compte tous les concepts presentés dans le chapitre précédent.

Le sous-ensemble de base de VHDL que nous avons choisi inclut les objets de type

VHDL	Expressions
`if c < 4 then x+5 else b+3 ;`	IF(c $<$ 4, x+5, b+3)
`if a+x > 10 then y ;`	IF(a+x $>$ 10, y, $undef$)

TAB. 2.2 – Exemple de transformations d'expressions

scalaire (entier, réel et booléen) et les expressions sur ces objets, l'affectation sequentielle, l'instruction *if then else*, la composition d'instructions séquentielles, l'affectation concurrente de signal, le processus avec une liste de sensibilité et la composition d'instructions concurrentes.

2.6.1 La transformation d'un sous-ensemble de base de VHDL

a) La transformation d'expressions sur des objets scalaires

Trans : $\mathcal{E}xpr \rightarrow \mathcal{E}xpr$

Les expressions VHDL, après l'application de la fonction **Trans**, restent identiques sauf pour les expressions *if* :

$$\textbf{Trans}(n) \triangleq n \qquad \textbf{Trans}(x) \triangleq x \qquad \textbf{Trans}((e)) \triangleq (\textbf{Trans}(e))$$

$$\textbf{Trans}(\text{op } e) \triangleq \text{op } \textbf{Trans}(e) \qquad \textbf{Trans}(e \text{ op } f) \triangleq \textbf{Trans}(e) \text{ op } \textbf{Trans}(f)$$

$$\textbf{Trans}(function\_call(arg_1, \ldots, arg_n)) \triangleq$$
$$function\_call(\textbf{Trans}(arg_1), \ldots, \textbf{Trans}(arg_n))$$

$$\textbf{Trans}(\textbf{if } b \textbf{ then } expression_1 \textbf{ endif}) \triangleq IF(\textbf{Trans}(b), \textbf{Trans}(expression_1), undef)$$

$$\textbf{Trans } (\textbf{if } b \textbf{ then } expression_1 \textbf{ else } expression_2 \textbf{ endif}) \triangleq$$
$$IF(\textbf{Trans}(b), \textbf{Trans}(expression_1), \textbf{Trans}(expression_2))$$

Les expressions *if* sont transformées dans des fonctions IF. Cette fonction est l'équivalent mathématique de l'instruction *if then else*. Si la condition b est vraie, alors la fonction IF retourne la valeur de la première expression, sinon la valeur de la deuxième expression. S'il n'existe pas de branchement *else*, la valeur de la fonction, dans le cas où la condition est fausse, n'est pas définie.

La fonction IF est définie dans [87] comme :

$$IF : \mathbb{B} \times \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$$

$$\text{IF } (true,\text{E},\text{F}) = \text{E} \qquad \text{IF } (false,\text{E},\text{F}) = \text{F}$$

Le tableau 2.2 présente quelques exemples de transformations d'expressions.

VHDL	Affectations parallèles
`s <= 0 ;`	$s := 0\,;$
`a := 1 ;`	$a := 1\,;$
`v <= if d > 10 then v+12 ;`	$v :=$ IF(d $>$ 10, v+12, $\widetilde{v}$)$\,;$

TAB. 2.3 – Exemple de transformation d'affectations des signaux et variables scalaires

b) La transformation de l'affectation séquentielle d'objets scalaires

Trans $(x <= expression) \triangleq x :=$ **Trans** $(expression)[undef/\widetilde{x}]$

L'affectation d'un signal est transformée dans une affectation parallèle ayant dans la partie droite la traduction de l'expression.

Nous avons vu dans la section précédente que les expressions *if* sont traduites vers des fonctions IF. Dans le cas particulier où l'expression *if* n'a pas de branchement *else* et sa condition est fausse, la fonction IF n'est pas définie. Si un objet est affecté avec une telle expression, la sémantique de l'affectation est que l'objet ne change pas sa valeur si la condition est fausse (il mémorise son ancienne valeur).

La mémorisation est marquée de manière différente pour la variable et le signal. Si une variable ne change pas, elle garde la valeur avant l'exécution de l'affectation. Pour un signal, du fait que l'affectation n'est pas immédiate, et que le changement de la valeur est effectué au prochain cycle de simulation, sa valeur est une prédiction. Nous devons alors faire une distinction entre la valeur du signal du cycle courant x et la valeur prédite du signal pour le cycle prochain que nous allons noter par $\widetilde{x}$. Donc, dans la règle de transformation, nous allons remplacer le symbole *undef* avec la valeur future prédite du signal avant affectation : $\widetilde{x}$.

Le tableau 2.3 présente quelques exemples de transformation des affectation d'objets scalaires.

c) La transformation d'instructions conditionnelles

Trans (**if** *condition* **then** Seq **endif**) $\triangleq$
$\|_{x \in \mathrm{W_{Seq}}}$ { x := IF (**Trans**(*condition*), $extract(x,$ **Trans**$(Seq))$,
$extract(x, \emptyset)$) }

Trans (**if** *condition* **then** $\mathrm{Seq_1}$**else** $\mathrm{Seq_2}$ **endif** $\triangleq$
$\|_{x \in \mathrm{W_{Seq_1}} \cup \mathrm{W_{Seq_2}}}$ { x := IF(**Trans**(*condition*), $extract(x,$ **Trans**$(\mathrm{Seq_2}))$,
$extract(x,$ **Trans**$(\mathrm{Seq_1}))$) }

VHDL	Affectations parallèles

`if cnt='0' then`
`  if bl="000000" then`
`  done <= '1';`
`  etat <= idle;`
`  else etat <= init;`
`  end if;`
`else etat <= cnt_reset;`
`end if;`
	etat :=
	IF (cnt = '0',
	IF (bl="000000", idle, init),
	cnt_reset)
	‖
	done :=
	IF (cnt='0'
	IF (bl="000000",
	'1', $\widetilde{done}$)
	$\widetilde{done}$)

TAB. 2.4 – Exemple de transformation de l'instruction *if then else*

La règle de transformation crée une seule affectation conditionnelle pour chaque variable écrite dans l'instruction *if then else* et les compose en parallèl.

Dans un premier temps, **Trans** est appelée récursivement sur les parties *condition*, $then-part$ et $else-part$ de l'instruction *if then else*, en considérant que la première est une expression booléenne, et les deux dernières sont des blocs d'instructions séquentielles (appartiennent au $\mathcal{S}eq$).

Dans un deuxième temps, le résultat de l'application de **Trans** où toutes les instructions sont des affectations parallèles, est donné comme entrée à l'opérateur *extract*.

Finalement, l'instruction est réécrite vers la fonction IF.

L'opération *extract* sélectionne à partir d'un ensemble d'affectations parallèles celle qui correspond à l'objet x et renvoie l'expression de la partie droite. Si l'objet n'est pas écrit par l'ensemble, alors sa valeur ne change pas, et l'opérateur renvoie x, si x est une variable ou $\widetilde{x}$ si x est un signal.

$$extract : (\mathcal{V} \cup \mathcal{S}) \times \mathbf{AP} \rightarrow \mathcal{E}xpr$$

L'opérateur a une définition structurelle :

- $extract(x,\ y := \mathrm{E}) = \begin{cases} \mathrm{E}, & \text{si } x = y \\ x, & \text{si } x \neq y \,\text{et}\, x \in \mathcal{V} \\ \widetilde{x}, & \text{si } x \neq y \,\text{et}\, x \in \mathcal{S} \end{cases}$
- $extract(x,\ \mathrm{S} \parallel \mathrm{Q}) = \begin{cases} extract(x, \mathrm{S}), & \text{si } x \in \mathrm{W_S} \\ extract(x, \mathrm{Q}), & \text{autrement} \end{cases}$

Dans le tableau 2.4, un bloc de la partie contrôle du circuit implémentant le SHA-1 est présenté, décrivant le comportement du composant dans l'état final (cnt_reset). Nous montrons une instruction VHDL *if then else* imbriquée, et les affectations parallèles correspondantes obtenues après l'application de la méthode d'extraction.

VHDL	Affectations parallèles

`a := b - cod ;`
`b <=0 ;`
`if (a = 0) then`
`  b <= cod + 2 ;`
`else cod := 5 ;`
`end if ;`
`if (b = 1) then`
`  cod := a ;`
`  state<=s1 ;`
`end if ;`
	a := b + cod
	‖
	b := IF (b - cod = 0, cod + 2, 0)
	‖
	cod := IF(b = 1, b - cod,
	IF(b - cod = 0, cod, 5))
	‖
	state := IF(b = 1, s1, $\widetilde{state}$)

TAB. 2.5 – Exemple de transformation d'un bloc d'instructions séquentielles

d) La transformation des blocs d'affectations séquentielles

La difficulté consiste à transformer plusieurs affectations du même signal sous des conditions successives et parfois non disjointes, en une seule affectation où toutes les conditions sont regroupées dans la partie droite.

$$\textbf{Trans}\ (\mathrm{Seq}_1\ ;\mathrm{Seq}_2) \triangleq \\ \|_{x \in \mathrm{W}_{\mathrm{Seq}_1} - \mathrm{W}_{\mathrm{Seq}_2}} \{x := \mathit{extract}(x, \textbf{Trans}_{\mathrm{Seq}_1}\} \\ \|\mathit{substitute}(\textbf{Trans}(\mathrm{Seq}_1), \textbf{Trans}(\mathrm{Seq}_2))$$

Après l'application de la fonction $\textbf{Trans}_{Seq}$ pour chaque instruction séquentielle, deux listes d'affectations sont produites. Les listes séquentielles sont transformées en parallélisme par la propagation des affectations du premier bloc Seq_1 dans le deuxième Seq_2.

Pour tout objet x écrit par Seq_1 et lu par Seq_2, la valeur du x dans Seq_2 est sa valeur après l'exécution de l'affectation correspondante de Seq_1. Ainsi, x est substitué en Seq_2 avec l'expression spécifiée par Seq_1. Pour les mêmes raisons liées à la sémantique, la substitution d'une variable est différente de celle d'un signal. Pour une variable, sa valeur courante v est remplacée, tandis que pour un signal s, sa valeur prédite $\widetilde{s}$ est remplacée. Cette transformation est réalisée par l'opérateur *substitute* qui retourne un ensemble d'affectations pour tous les objets écrits par Seq_2. Les affectations des objets écrits seulement par le premier bloc ne sont pas modifiées. Les affectations doubles sont éliminées durant le processus : seule l'affectation du dernier bloc est prise en compte, après que les substitutions correspondantes ont été réalisées.

L'opération *substitute*(S,Q) propage les effets de S à l'intérieur de Q.

$$\mathit{substitute} : \textbf{AP} \times \textbf{AP} \rightarrow \textbf{AP}$$

L'opération est définie structurellement sur l'ensemble des affectations parallèles.

VHDL	Affectations parallèles
`sxor <= ern xor ki ;`	sxor := IF(*Event*(ern, ki), ern xor ki, sxor)

TAB. 2.6 – Exemple de transformation d'affectation concurrente de signal de type scalaire

- Le changement d'une variable x se propage dans les expressions qui la lisent :

$$substitute(x := \mathrm{E},\ y := \mathrm{F}) = y := \mathrm{F}[x/\mathrm{E}]\ ,\ \text{si}\ x \in \mathcal{V}$$

- Le changement d'un signal x est propagé seulement dans les expressions qui lisent sa valeur mémorisante :

$$substitute(x := \mathrm{E},\ y := \mathrm{F}) = y := \mathrm{F}[\widetilde{x}\ /\ \mathrm{E}]\ ,\ \text{si}\ x \in \mathcal{S}$$

- Le changement d'un objet n'influence pas les expressions qui ne le lisent pas :

$$substitute(\mathrm{S},\ \mathrm{Q}) = \mathrm{Q}\ \text{si}\ \mathrm{W_S} \cap \mathrm{R_Q} = \emptyset$$

- La propagation d'une affectation S dans une composition parallèle est équivalente avec la composition parallèle de propagations de l'affectation S dans chacune des affectations S_i de la composition :

$$substitute(\mathrm{S},\ \mathrm{S}_1 \parallel \mathrm{S}_2 \cdots \parallel \mathrm{S}_n) = \parallel_{i \in 1,n} substitute(\mathrm{S}, \mathrm{S}_i)$$

  Puisque des variables écrites par S_i peuvent être utilisées par S_j, $j \leq i$, il n'est pas possible de propager d'abord S_j et après S_i. Ainsi la propagation d'une composition parallèle dans une affectation est une substitution parallèle :

$$substitute(\mathrm{S}_1 \parallel \mathrm{S}_2 \cdots \parallel \mathrm{S}_n,\ y := \mathrm{F}) := y := \mathrm{F}\ [x/\mathrm{E}, \widetilde{z}\ /\mathrm{G}]$$

$\forall x \in \cup_{i \in 1,n} \mathrm{W}_{\mathrm{S}_i} \cap \mathcal{V}, \mathrm{S}_i : x := \mathrm{E}$ et $\forall z \in \cup_{j \in 1,n} \mathrm{W}_{\mathrm{S}_j} \cap \mathcal{S}, \mathrm{S}_j : z := \mathrm{G}$

La transformation d'un bloc d'instructions séquentielles est présentée dans le tableau 2.5.

e) La transformation de l'affectation concurrente de signal de type scalaire

Trans $(x <= expression) \triangleq$
$x :=$ IF $(Event\ (Sensitivity\ (expression))$,
Trans$(expression)[undef/x],\ x)$

Ce cas correspond à une affectation simple de signal avec une expression. Conformément à la sémantique VHDL, l'affectation est effectuée sans contraintes dans la phase d'initialisation. Par la suite, elle est effectuée seulement s'il existe un événement sur un ou plusieurs signaux présents dans l'expression. *Sensitivity* renvoie la liste de signaux d'une expression, c'est à dire la liste de sensibilité pour la cible. La fonction *Event* prend comme

VHDL	Affectations parallèles

`process(clk)`
`begin`
` if (clk'event and clk='1') then`
`  if(ctrl_buf='1') then`
`  res<=c ;`
`  end if ;`
` end if ;`
`end process ;`
	res := IF (*Event*(clk),
	IF(*Event*(clk) and clk='1',
	IF(ctrl_buf='1', c, res),
	res),
	res)

TAB. 2.7 – Exemple de transformation de processus

entrée une liste de signaux et vérifie s'il y a eu des évènements sur les éléments de la liste. La valeur retournée est booléenne.

Si le signal est affecté avec une expression *if then* (sans le branchement *else*), dans le branchement correspondant le signal garde sa valeur ancienne. Puisque l'instruction est concurrente, cette valeur est la valeur du signal pour le cycle de simulation courant, donc les symboles *undef* sont remplacés avec la valeur du signal x (tableau 2.6).

f) La transformation du processus avec une liste de sensibilité

Trans (*label* : **process** (*sensitivity_list*) Decl **begin** Seq **end process**) $\triangleq$
Trans (**if** *Event*(sensitivity_list) **then** Seq **endif**)

Il existe deux types de processus en VHDL : soit avec une liste de sensibilité, soit qui contiennent des instructions de synchronisation *wait*. Ce cas traite le processus concurrent avec une liste de sensibilité. La fonction **Trans** est appliquée sur le bloc d'instructions séquentielles du processus, en distribuant la liste de sensibilité sur toutes les instructions (tableau 2.7). Les noms des objets définis localement sont préfixés avec l'étiquette du processus, afin de garantir une appellation unique.

g) La transformation des blocs d'affectations concurrentes

Trans (Conc_1 ; Conc_2) $\triangleq$ *resolved*(**Trans** (Conc_1), **Trans** (Conc_2))$[\widetilde{x}/x]$

La composition concurrente d'instructions est transformée en une affectation parallèle de la manière suivante : dans en premier temps, chaque bloc d'instructions concurrentes est transformé dans une affectation parallèle. Ensuite, la fonction *resolved* est appliquée afin de résoudre les éventuels conflits d'écriture. Finalement, les occurences de $\widetilde{x}$ sont remplacées

par x, puisque le résultat de calcul pour un cycle de simulation est calculé, et la valeur future presumée au debut du cycle de simulation est la valeur courante.

La fonction *resolved* retourne l'ensemble des affectations correspondantes pour les signaux et les variables affectées dans les deux blocs d'instructions parallèles.

$$resolved : \mathbf{AP} \times \mathbf{AP} \rightarrow \mathbf{AP}$$

$$\begin{aligned} resolved(S,T) = & \|_{x \in W_S \cup W_T - W_S \cap W_T} \{x := E\} \cup \\ & \|_{x \in W_S \cap W_T \cap \mathcal{S}} \{x := Res(x,E,F), x := E \in S \text{ et } x := F \in T\} \end{aligned}$$

Si les deux blocs ont écrit les mêmes variables, $W_S \cap W_T \cap \mathcal{V} \neq \emptyset$, alors la description a des variables partagées et donc ne correspond pas à notre sous-ensemble VHDL. Si les deux blocs ont écrit les mêmes signaux, la fonction *resolved* regarde si les positions écrites sont les mêmes. Si oui, des fonctions de résolution correspondantes sont appliquées. Si les signaux n'ont pas un type résolu, alors il y a une erreur dans la description.

2.6.2 La transformation des objets composés

Nous considérons un objet de type composé (tableau ou structure) comme formant un tout. Ainsi, si une partie de l'objet est écrite, nous considérons que l'objet lui même est modifié. De même, si une partie de l'objet est lue, nous considérons que l'objet lui même est lu.

Une partie d'un l'objet de type composé est désignée par un identificateur VHDL défini récursivement comme suit :

- le nom de l'objet désigne l'objet lui-même ;
- $id.nom$ désigne le champs nom de l'objet id de type enregistrement ;
- $id(n)$ désigne l'élément n du tableau id ;
- $id(a_1 \text{ dir } a_2)$ désigne la partie du tableau id délimitée par les éléments a_1 et a_2.

Nous définissons la fonction *pos* qui prend en paramètre un identifiant, et retourne la position de l'élément désigné dans l'objet. Nous répresentons une position sous forme de liste définie comme suit :

- nil est la position vide.
- si a est un entier et p est une position, alors (a,p) est une position.
- si a et b sont des entiers alors (to,a,b) et $(downto,a,b)$ sont des positions.

Soit Id l'ensemble d'identificateurs et Pos l'ensemble de positions. La fonction *pos* est définie récursivement sur la structure d'identificateurs :

$$pos : Id \rightarrow Pos$$

$pos(x) = nil$
$pos(id.var) = pos(id) \& (pos_{var}(var, \tau(id)))$
$pos(id(a_1,...,a_n)) = pos(id)\ \&(a_1,...,a_n)$

$pos(id(a_1 \text{ to } a_2)) = pos(id)$ & ((to, a_1, a_2))
$pos(id(a_1 \text{ downto } a_2))= pos(id)$ & ((downto, a_1, a_2))

Dans la définition de la fonction *pos*, τ retourne le type d'un identificateur et pos_{var} retourne la position d'un champ dans un type enregistrement.

Soit le type enregistrement T=(var_0 : T_0 ; ... var_n : T_n) où var_i est le nom et T_i est le type du champ i de la structure, $0 \leq i \leq n$. A chaque nom de champ nous associons une valeur qui donne sa position dans le type : $pos_{var}(var_i, T)$=i.

Soit le code VHDL suivant :

```
type color is array (0 to 7) of BIT;
type pixel is
 record
 red : color;
 green : color;
 blue : color;
 end record;
type line is array (0 to 31) of pixel;
type image is array (0 to 31) of line;
```

et la déclaration :

```
variable a : image;
```

Pour l'identificateur `a(2,5).red`, la fonction *pos* retourne la liste (2,5,0), car le type de `a(2,5)` est `pixel` et pos_{var}(`red`,`pixel`)=0.

Pour l'identificateur `a(7)(2 to 5)`, la fonction *pos* retourne la liste (7,(to, 2, 5)).

Maintenant traitons la transformation d'un objet de type composé dont une partie identifiée par *id* est lue.

Trans$(id) \triangleq nth(pos(id), W_{id})$

L'identificateur est remplacé par l'appel de la fonction *nth* qui prend en argument une position p et un objet $x \in \mathcal{S} \cup \mathcal{V}$ et retourne la valeur qui se trouve sur la position p dans l'objet : $x(p)$.

$$nth : Pos \times \mathcal{T} \rightarrow \mathcal{T}$$

La fonction $nth(p, x)$ est définie de la manière suivante :

– si la position est la liste vide, la fonction retourne la valeur de l'objet x :

$$nth(p, x) = x \text{ si } p = nil$$

VHDL	Expressions
`b(2,6)+5*a(2,5) ;`	nth((2,6),b) + 5* nth((2,5),a)
`d(7)(2 to 5) & d(7)(0 to 1) ;`	nth((7, (to 2 5)), d) & nth((7,(to 0 1)),d) ;
`y(7,23) or en_write(a,c(0)) ;`	nth((7, 23), y) or en_write(a, nth((0),c)) ;
`if a(3)< 4 then d else b+3 ;`	IF(nth((3), a)<4, d, b+3)

TAB. 2.8 – Exemple de transformations d'expressions avec des objets composées

– sinon, la fonction retourne la valeur qui se trouve sur la position p de la valeur de l'objet x :

$$nth(p, x) = x(p) \text{ si } p \neq nil$$

Le tableau 2.8 présente quelques exemples de transformations d'expressions avec des objets composés.

Maintenant traitons le cas d'écriture d'une partie d'un élément de type composé.

2.6.3 La transformation de l'affectation d'un objet de type composé

Quand un des éléments d'un tableau ou structure est affecté, nous considérons que l'objet lui-même a été affecté et nous générons une affectation parallèle pour l'objet dans sa totalité. (Tableau 2.9).

Traitons d'abord le cas de l'affectation du signal.

Trans ($id <= expression$) $\triangleq$
$\quad \mathrm{W}_{id} := replace\ ((pos(id),$ **Trans** $(expression)[undef/\widetilde{\mathrm{W}}_{id}]), \widetilde{\mathrm{W}}_{id})$

Soit le signal s de type tableau qui a au moins deux dimensions. Si le mot i de signal est affecté avec une nouvelle expression E, ce qui nous intéresse est que dans le signal s, à la position i, la valeur a changé, tandis que pour les autres positions les valeurs sont memorisées : $replace$ ((i,E), $\widetilde{s}$).

La fonction $replace$ modélise la modification de la valeur d'un objet après l'exécution d'une affectation le concernant. La fonction prend comme entrées une liste de couples ($position$, $expression$), et un objet, et retourne la valeur de l'objet après les changements définis par la liste. Dans le cas particulier des objets de type de base, la position est évaluée à nil. Le résultat de la fonction est dans ce cas l'expression avec laquelle l'objet est affecté : $replace\ ((nil, expression), x) = expression$.

Soit $2^{Pos \times \mathcal{E}xpr}$ l'ensemble des listes de couples ($position$, $expression$), où $position \in Pos$ et $expression \in \mathcal{E}xpr$.

VHDL	Affectations parallèles
`s(i) <= 0 ;`	$s := replace\ ((\mathrm{i}, 0), \widetilde{s})$;
`a(2,5) := 1 ;`	$a := replace\ (((2, 5), 1), a)$;
`b(4 to 8) <= c(9 downto 5) ;`	$b := replace\ (((\mathrm{to}, 4, 8), nth((\mathrm{downto}, 9, 5), \mathrm{c})), \widetilde{b})$;

TAB. 2.9 – Exemple de transformation d'affectations des signaux et variables

$$replace : 2^{Pos \times \mathcal{E}xpr} \times \mathcal{T} \rightarrow \mathcal{T}'$$

La fonction $replace(L, x)$ est définie de la manière suivante :

– si la liste des positions L est vide, la fonction retourne la valeur de l'objet :

$$replace(L, x) = x \text{ si } L = nil$$

– si la liste L n'est pas vide, alors elle a la forme ((p,F), L). Si la position p est la liste vide, l'objet x est remplacé par l'expression F.

$$replace([(p, F), L], x) = F \text{ si } p = nil$$

– si la position p est une liste non vide, la fonction retourne la valeur de l'objet x, après que tous les éléments qui se trouvent dans les positions p de la liste L sont remplacés par les expressions F correspondantes. Si p est une position de type (dir, a, b), où $dir \in \{to, downto\}$, alors le segment (a, b) de l'objet x est remplacé par F. Le remplacement est réalisé de l'intérieur vers l'extérieur :

$$replace([(p, F), L], x) = x', \text{ où}$$

$$x'(k) = \begin{cases} F, & \text{si } k = p \\ F(k), & \text{si } k \in p \\ replace(L, x)(k) & \text{autrement} \end{cases}$$

et $k \in \mathbb{Z}^n$. n est la longueur de la position p (la position étant une liste).

L'affectation d'une variable est modélisée de manière similaire, le symbole $undef$ étant remplacé par la valeur courante de la variable : W_{id}.

Trans $(id := expression) \triangleq$
$\mathrm{W}_{id} := replace((pos(id), \mathbf{Trans}(expression)[undef/\mathrm{W}_{id}]), \mathrm{W}_{id})$

2.6.4 La transformation d'instructions de boucle

L'instruction *while* et transformée en une affectation parallèle. Pour chaque objet x appartenant à l'ensemble $\mathrm{W}_{\mathrm{Seq}}$, une affectation est définie.

Trans (**while** *cond* **do** Seq **end do**) $\triangleq$
$\|_{x \in \mathrm{W_{Seq}} \cap \mathcal{V}} \{x := IF(cond, while\_x(x, \mathrm{R}_{E_x} \cup \mathrm{R}_{cond}), x)\}$
$\|_{x \in \mathrm{W_{Seq}} \cap \mathcal{S}} \{x := IF(cond, while\_x(x, \mathrm{R}_{E_x} \cup \mathrm{R}_{cond}), \widetilde{x})\}$

où $while\_x(x, x_1, \ldots, x_k) = IF(cond, while\_x(E_x[\widetilde{x}/x], F_{x_1}, \ldots, F_{x_k}), x)$

et $E_x = extract(x, \mathbf{Trans}(\mathrm{Seq}))$ et
$\mathrm{R}_{E_x} \cup \mathrm{R}_{cond} = \{x_1, \ldots, x_k\}$ et
F_{x_i} est E_{x_i} si $x_i \in \mathcal{V} \cap \mathrm{W_{Seq}} \cap (\mathrm{R}_{E_x} \cup \mathrm{R}_{cond})$, et x_i autrement.

L'expression de l'affectation est un appel de la fonction IF. Si la condition est vraie, la valeur de l'objet après l'exécution de la boucle est retournée, sinon la valeur calculée avant l'instruction est retournée, c'est à dire la valeur mémorisée (x pour les variables et $\widetilde{x}$ pour les signaux).

Dans un premier temps, les instructions séquentielles qui forment le corps de la boucle sont transformées dans des affectations parallèles, une pour chaque objet (variable ou signal) écrit par *Seq*. Pour chaque objet, son affectation décrit comment calculer la valeur de l'objet après une itération de la boucle. Pour connaître la valeur de l'objet après l'exécution de la boucle, il faut définir une fonction récursive : $while\_x$.

$while\_x$ prend comme arguments l'objet à calculer et les objets qui sont nécessaires pour calculer la condition *cond* et son expression. Si la condition est fausse, la valeur de l'objet est retournée, si la condition est vraie, la fonction $while\_x$ est appelée récursivement. Dans l'appel récursif, les variables sont mises à jour avec les expressions E_{x_i} calculées par le corps de la boucle. Les signaux restent les mêmes car leurs valeurs seront modifiées seulement au cycle prochain. La valeur de l'objet est recalculée en tenant compte des nouvelles valeurs de variables. Dans le cas des signaux, la valeur mémorisante $\widetilde{x}$ est mise à jour avec la valeur calculée à l'itération précédente.

L'instruction *for* est un cas particulier de l'instruction *while*. La condition d'arrêt est $i \leq a_2$, en sachant que i est initialisé avec a_1.

Trans (**for** i **from** a_1 **to** a_2 **do** Seq **end for**) $\triangleq$
$\|_{x \in \mathrm{W_{Seq}} \cap \mathcal{V}} \{x := IF(a_1 \leq a_2, for\_x(x, a_1, \mathrm{R}_{E_x} \cup \mathrm{R}_{a_2}), x)\}$
$\|_{x \in \mathrm{W_{Seq}} \cap \mathcal{S}} \{x := IF(a_1 \leq a_2, for\_x(x, a_1, \mathrm{R}_{E_x} \cup \mathrm{R}_{a_2}), \widetilde{x})\}$

où $for\_x(x, i, x_1, \ldots, x_k) = IF(i \leq a_2, for\_x(E_x[\widetilde{x}/x], i+1, F_{x_1}, \ldots, F_{x_k}), x)$.

Plusieurs exemples de transformation de boucle sont présentés dans le Tableau 2.10.

VHDL	Affectations parallèles

`for i in 1 to n loop`
`  if ((addr >= 0) and`
`  (addr < size)) then`
`  mem(addr) <=`
`  to_unsigned(val,8) ;`
`  end if ;`
`  addr := addr+1 ;`
`end loop ;`
	mem :=IF (1 ≤ n,
	for_{mem}(mem, 1, addr, val),
	$\widetilde{mem}$)
	‖
	$address$:=IF (1 ≤ n,
	$for_{address}$(addr,1), addr)

où for_{mem}(mem, i, addr, val)=
IF (i ≤ n, for_{mem} (IF ((addr >=0) and (addr<size),
replace ((addr, to_unsigned(val,8)), $\widetilde{mem}$), $\widetilde{mem}$),
i+1, addr+1, val),
mem)
et $for_{address}$ (addr, i)= IF(i ≤ n, $for_{address}$(addr+1, i+1), addr)

TAB. 2.10 – Exemple de transformation d'une boucle

2.6.5 La transformation des déclarations

Trans : $\mathcal{D}ecl \rightarrow$ **AP**

Les déclarations de type et de type d'objet ne sont pas modifiées. Par contre l'initialisation d'objet dans une déclaration est transformée en une affectation d'objet :

1. **Trans**(**signal** *name* : type :=val) $\triangleq$ *name* := **Trans**(val)
   Trans(**variable** *name* : type :=val) $\triangleq$ *name* := **Trans**(val)
   Trans(**constant** *name* : type :=val) $\triangleq$ *name* := **Trans**(val)

2. **Trans** (type **function** *name* (params) Dec Seq **return** *expression*) $\triangleq$
   name (params) = $expression[x/extract(x, \textbf{Trans}(\text{Decl ; Seq}))]$
$\forall x \in \text{R}_e xpression \cap \text{W}_{\text{Seq}}$

La déclaration de fonction est transformée dans une définition de fonction.
$name : T_{x_1} \times \ldots \times T_{x_n} \rightarrow$ type

où T_{x_i} est le type de paramètre x_i.

Une fonction est transformée dans une expression. Dans un premier temps, les déclarations sont transformées afin de pouvoir prendre en compte d'éventuelles initialisations

de variables. Ensuite, l'ensemble d'instructions séquentielles qui forment le corps de la fonction est transformé dans des affectations parallèles conformément aux règles décrites auparavant. Finalement, dans l'expression à retourner, toutes les variables modifiées par le corps de la fonction sont remplacées par leurs expressions correspondantes. Si la fonction est récursive, alors l'appel récursif se retrouve dans l'expression (Tableau 2.11).

Trans (**procedure** *name* (params) Decl ; Seq) $\triangleq$
name (params) = $\|_{x \in params \cap \mathrm{W_{Seq}}} \{x := extract(x, \mathbf{Trans}(\mathrm{Decl}\,;\,\mathrm{Seq}))\}$

La déclaration de procédure est transformée en une définition de fonction :
$name : T_{x_1} \times \ldots \times T_{x_n} \rightarrow T_{x_1} \times \ldots \times T_{x_p}$

où T_{x_i} est le type de paramètre x_i et $p \leq n$, car les paramètres ne sont pas tous modifiés par la procédure.

Le corps de la procédure est transformé en une affectation parallèle, où sont représentés seulement les paramètres de la procédure qui sont écrits par Seq. Les variables internes disparaissent, car leur comportement a déjà été pris en compte dans le calcul des affectations par la règle de transformation de la composition séquentielle. Si la procédure contient des instructions *wait*, alors son corps n'est pas transformé. Dans ce cas, l'appel de procédure est remplacé par le corps de la procédure lui-même, après le remplacement des paramètres de la procédure avec les arguments de l'appel.

Tableau 2.11 présente un exemple de tranformation de procédure.

La transformation de la configuration de composant est discutée dans la section suivante.

2.6.6 La transformation d'affectation concurrente d'objet composé

La fonction **Trans** est appliquée sur une affectation concurrente d'objet composé et produit une d'affectation parallèle pour le signal affecté. Même si seulement une partie du signal est affectée, l'affectation est réalisée s'il existe un événement sur un des signaux de la liste de sensibilité de la cible. Par contre, dans ce cas, seulement les éléments sur les positions affectées sont modifiés.

Trans ($id <= expression$) $\triangleq$
W_{id} := IF ($Event$ ($Sensitivity$ ($expression$)),
$replace((pos(id), \mathbf{Trans}(expression)[undef/\mathrm{W}_{id}]), \widetilde{\mathrm{W}}_{id})$,
W_{id})

2.6.7 La transformation d'un processus avec plusieurs *wait*

L'instruction *wait* a la forme *wait on* liste *until* condition. Si l'instruction *wait* n'a pas la clause *on* alors *liste* est la liste des signaux de la *condition* ($Sensitivity(c)$). S'il

VHDL	Affectations parallèles

`function adressage2`
`(count : in bit_vector(3`
`downto 0) ;`
`t :in integer)`
`return bit_vector is`
`variable X : bit_vector(3`
`downto 0) ;`
`begin`
`X :=count ;`
`if t=18 then X :=count ;`
`elsif t=19 then X :=X+2 ;`
`elsif t=20 then X :=X+8 ;`
`elsif t=21 then X :=X+13 ;`
`end if ;`
`return X ;`
`end adressage2 ;`
	adressage2(count,t) =
	IF(t=18, count
	IF(t=19, count+2,
	IF(t=20, count+8,
	IF(t=21, count+13, count))))

`procedure min_max(a,b : in`
`integer ; variable c,d : out`
`integer) is`
`begin`
`if a<b then`
`c :=a ; d :=b ;`
`else`
`c :=b ; d :=a ;`
`end if ;`
`end min_max ;`
	min (a, b, c, d) =
	c := IF (a < b, a ,b)
	‖
	d := IF (a < b, b ,a)

TAB. 2.11 – Exemple de transformation des sous-programmes

manque la clause *until*, alors *condition* est la constante *true*. Une instruction *wait* introduit un point de synchronisation. Par conséquent, nous définissons un point de synchronisation comme un triplet (i, L, C) où i est une étiquette, L est une liste de signaux et C est une condition.

La sémantique d'un processus est définie de la manière suivante : tant qu'il n'y a pas d'évènements sur les signaux d'attente dans le point de synchronisation, ou que la condition d'attente n'est pas remplie, le processus reste suspendu. Si, au contraire, il y a des évènements sur les signaux d'attente et que la condition d'attente est vraie, alors le processus est actif et l'execution continue avec l'instruction qui suit après le point de synchronisation. Le successeur de la dernière instruction du processus, est la première instruction du processus. Ceci est dû à la sémantique d'exécution des processus, qui stipule que le processus est exécuté en boucle.

Un processus avec plusieurs instructions *wait* peut être réécrit en un processus équivalent avec une liste de sensibilité.

Trans (*label* : **process** Decl **begin** Seq **end process**) $\triangleq$
  Trans (*label* : **process** (*sensitivity_list*)
    Decl ; **variable** pw : integer := 1 ;
    begin $Modify$ (Seq) **end process**)

sensitivity_list est l'ensemble des signaux qui apparaissent dans les expressions de l'instruction *wait*.

Le nouveau processus a une nouvelle variable *pw* qui est initialisée à 1. Cette variable, propre à chaque processus, a comme valeur courante l'étiquette de l'instruction *wait* qui a suspendu le processus.

La fonction $Modify$ transforme un bloc séquentiel avec des instructions *wait* en une instruction *if then else* comme suit :

$Modify$ (Seq_0 ; wait on L_1 until C_1 ; Seq_1 ; ... wait on L_n until C_n ; Seq_n) $\triangleq$
  (**if** pw=1 and $Event(Sensitivity(L_1)$ and C_1
    then Seq_1 ; pw :=2 ;
    else if pw=2 and $Event(Sensitivity(L_2)$ and C_2
      then Seq_2 ; pw :=3 ;
      ...
      else if pw=n and $Event(Sensitivity(L_n)$ and C_n
        then Seq_n ; Seq_0 ; pw :=1 ; **endif**)

Soit n le nombre d'instructions *wait* d'un processus (n est calculable car un programme est fini). Si le processus se trouve dans un des points de synchronisation ($1 \leq pw \leq n$), et des évènements ont eu lieu sur les signaux de synchronisation et la condition d'attente est remplie, alors le bloc d'instructions séquentielles est exécuté jusqu'à la rencontre d'une nouvelle instruction *wait*. Puisque le processus a un nouveau point de synchronisation, la valeur de *pw* est mise à jour.

2.6.8 La transformation de composants

Tout composant instancié doit être configuré. La configuration de composant est transformée en une fonction qui a le nom de l'architecture, préfixé de celui de l'entité et de celui du composant.

Trans (*name* (params) Ent ; Arch) $\triangleq$
$name\_Ent\_Arch$ (params $\cup \mathcal{L}ocals$) = $\|_{x \in \mathcal{L}ocals}\{x := extract(x, \textbf{Trans}(\text{Conc}))\}$

$name\_Ent\_Arch : T_{x_1} \times \ldots \times T_{x_n} \rightarrow T_{x_1} \times \ldots \times T_{x_p}$

où T_{x_i} est le type de x_i et $p \leq n$, car les entrées ne sont pas modifiées par le composant.

L'architecture est transformée en une affectation parallèle conformément aux règles définies pour les instructions concurrentes (la section suivante). Le résultat de la transformation contient une affectation pour chaque objet écrit dans le composant. Pourtant, certains objets ne sont ni des entrées, ni des sorties, ni des éléments mémorisants du composant, donc il n'y a aucun intérêt à les garder.

Soit $\mathcal{O}ut$ l'ensemble des sorties modifiées par le composant. Pour chaque sortie x de $\mathcal{O}ut$ nous allons calculer l'ensemble d'objets nécessaires pour son calcul.

Soit E_x=$extract(x, Trans(Conc))$ l'expression affectée à x, où $Conc$ est le bloc d'instructions correspondant à l'architecture, alors nous définissons $\mathcal{D}_0$:=$\cup_{x \in \mathcal{O}ut} R_{E_x}$. Nous définissons la suite $\mathcal{D}_{n+1} = \cup_{x \in \mathcal{D}_n} R_{E_x} \cup \mathcal{D}_n$. Le plus petit ensemble avec cette propriété, $\mathcal{L}ocals$, est l'ensemble des paramètres finaux de la fonction attachée au composant. Les paramétres sont des objets mémorisants du composant. Afin de garantir une appellation unique, leur nom est préfixé avec le nom du composant.

Finalement, le corps de la fonction est l'ensemble des affectations correspondant aux objets de $\mathcal{L}ocals$.

L'Annexe 6 présente les tableaux récapitulatifs des transformations définies dans les sections précédentes.

2.6.9 La construction d'un SER

L'obtention d'un SER à partir des affectations parallèles est directe. Soit C une description VHDL d'un circuit. Elle est de la forme C=(Ent, $Arch$), où l'entité Ent est un ensemble de déclarations (de paquetage, types, etc) et l'architecture $Arch$ a une partie déclaration et une partie instructions concurrentes $Conc$.

Soit AP le résultat de **Trans**(Conc) et $Decl$ l'ensemble de déclarations de la description. Chaque affectation de AP est transformée en une équation récurrente comme suit : $x := E$ devient $x(t+1) = E_x[y/y(t)]$, $\forall y \in \mathrm{R}_{E_x}$. La partie gauche de l'affectation représente le résultat d'un cycle de simulation, il est donc la valeur de l'objet à l'instant futur, t+1. Dans la partie droite de l'affectation, l'expression est calculée en fonction de la valeur courante de l'objet, c'est à dire la valeur à l'instant t.

Le SER correspondant à la description C, SER_C, est

$$SER_C = \langle \mathcal{I}nput, \mathcal{L}ocals, \mathcal{O}ut, Init, \{E_x : x(t+1) = f_x(y(t), \ldots, z(t-1))\}_{x \in \mathcal{L}ocals \cup \mathcal{O}ut}, \mathcal{D}ecl \rangle$$

$\mathcal{I}nput$ est l'ensemble des entrées décrites dans l'entité Ent, $\mathcal{L}ocals$ est l'ensemble des objets internes dont dépend le calcul des sorties du composant (aussi appelé cône d'influence). $\mathcal{L}ocals$ est calculé de la même manière que celle présentée pour un composant : il est la fermeture transitive de la suite : $\mathcal{D}_0 = \cup_{x \in \mathcal{O}ut} R_{E_x}$, $\mathcal{D}_{n+1} := \cup_{x \in \mathcal{D}_n} R_{E_x} \cup \mathcal{D}_n$. $\mathcal{D}ecl$ sont les déclarations du système.

$Init$ est l'ensemble des valeurs initiales des objets internes et de sorties, $\mathcal{L}ocals \cup \mathcal{O}ut$. Ceux-ci sont obtenus conformément à la sémantique de VHDL, à partir d'initialisations dans des déclarations, et en exécutant une fois les instructions concurrentes. Donc

$$Init = \{x(0) = I_x[y/y(0)],\ \forall y \in \mathrm{R}_{E_x}\}_{x \in \mathcal{L}ocals} \cup \mathcal{O}ut.$$

$I_x = extract(x, \mathbf{Trans}_{Init}(Conc))$ si $x \in \mathrm{W}_{\mathbf{Trans}_{Init}(Conc)}$, et $x(0)$, autrement.
La fonction $\mathbf{Trans}_{Init}$ calcule les valeurs initiales des objets d'une description.

$\mathbf{Trans}_{Init} : \mathcal{C}onc \rightarrow \mathbf{AP}$

1. a) $\mathbf{Trans}_{Init}$ $(x <= expression) \triangleq$
$x := \mathbf{Trans}(expression)[undef/x]$

1. b) $\mathbf{Trans}_{Init}$ $(id <= expression) \triangleq$
$\mathrm{W}_{id} := replace((pos(id), \mathbf{Trans}(expression)[undef/\mathrm{W}_{id}]), \widetilde{\mathrm{W}}_{id})$

2. a) $\mathbf{Trans}_{Init}$ ($label$: **process** ($sensitivity\_list$) Decl **begin** Seq **end process**) $\triangleq$
Trans (Decl ; Seq)

2. b) $\mathbf{Trans}_{Init}$ ($label$: **process** Decl **begin** Seq **end process**) $\triangleq$
$\mathbf{Trans}_{Init}$ ($label$: **process** ($sensitivity\_list$)
Decl ; **variable** pw : integer := 1 ;
begin $Initial$ (Seq) **end process**)

$Initial$ (Seq_0 ; wait C_1 ; Seq_1 ; ... wait C_n ; Seq_n) $\triangleq$ Seq_0

3. $\mathbf{Trans}_{Init}$ (Conc_1 | Conc_2) $\triangleq$ $resolved($ $\mathbf{Trans}_{Init}$ (Conc_1), $\mathbf{Trans}_{Init}$ (Conc_2))$[\widetilde{x}/x]$

Si la description est hiérarchique, alors le SER obtenu est hiérarchique.

Définition 13 (Système d'équations récurrentes hiérarchique) *Un système d'équations récurrentes hiérarchique est un n-uplet :*

$$SER = \langle \mathcal{I}nput, \mathcal{L}ocals, \mathcal{O}ut, Init, \{E_x : x(t+1) = f_x(y(t), \ldots, z(t-1))\}_{x \in \mathcal{L}ocals \cup \mathcal{O}ut}, \mathcal{D}ecl, Comp \rangle$$

où Comp est l'ensemble des composants du système. Chaque composant est un SER.

2.7 Simulation symbolique d'un SER

Dans les sections précédentes, nous avons montré comment obtenir un système d'équations récurrentes à partir d'une description VHDL. Les équations décrivent le comportement du circuit entre deux points de synchronisation. Ceci correspond à la phase d'évaluation de l'algorithme de simulation VHDL. Par la suite, nous montrons comment intégrer cette étape dans l'algorithme complet de VHDL.

Nous présentons une version simplifiée de l'algorithme de simulation VHDL tel qu'il est présenté dans la norme [59] :

L'exécution d'un modèle est composée d'une phase d'initialisation suivie d'une phase répétitive d'exécution d'instructions concurrentes du modèle. Chaque itération est un cycle de simulation. Dans chaque cycle, toutes les valeurs des signaux du modèle sont calculées. Si, comme résultat de ce calcul, un événement est déclenché pour un signal, les instructions sensibles à ce signal sont exécutées à nouveau dans un nouveau cycle de simulation sans avancer le temps.

Au début de l'initialisation, le temps courant t_c est mis à 0. La phase d'initialisation est la succession des pas suivants :

- la valeur courante de tous les objets déclarés est initialisée avec la valeur spécifiée dans la déclaration. Si aucune valeur n'est spécifiée, l'objet est initialisé avec la plus petite valeur de son type. Pour les signaux, on considère que leur valeur courante est égale à leur valeur précédente.
- tous les processus sont exécutés jusqu'à la rencontre d'une instruction *wait*.
- Le temps du prochain cycle de simulation t_n est calculé d'après les règles du pas e) de l'algorithme suivant.

Un cycle de simulation est constitué des pas suivants :

a) Le temps courant t_c prend la valeur du t_n. La simulation est complète quand t_n est égal au temps final de simulation T (fixé par l'utilisateur), et il n'existe plus d'événement sur des signaux ou des instructions concurrentes à exécuter au temps t_n.

b) Chaque signal du modèle est mis à jour, ce qui peut déclencher des événements.

c) Pour toute instruction concurrente *Conc*, si *Conc* est sensible à un signal s, et un événement a été déclenché sur s, alors l'instruction est exécutée.

d) Chaque instruction concurrente à exécuter dans le cycle courant est exécutée jusqu'à la rencontre d'une instruction *wait*. Après, son exécution est suspendue.

e) Quand toutes les instructions sont suspendues, le temps du prochain cycle de simulation t_n est calculé comme étant le plus petit parmi :

- le prochain temps qu'un signal devient actif
- le prochain temps qu'une instruction doit s'exécuter
- le temps de simulation T

Si t_n est égal à t_c alors le prochain cycle de simulation est un cycle delta.

Cette description de l'algorithme met en évidence deux notions : le temps de simulation et le cycle de simulation.

Notre sous-ensemble VHDL ne prend pas en compte les variables de type TIME et la clause $after$, ce qui induit plusieurs conséquences :

- un cycle de simulation a lieu dans une unité abstraite de temps ou dans 0 unité de temps s'il s'agit d'un cycle delta.

- si suite à l'exécution d'instructions concurrentes, la valeur d'un signal change, le changement est réalisé seulement au même instant de simulation.

- les instructions ne sont jamais retardées pour une période de temps. Par défaut, conformément à la sémantique de simulation VHDL, le temps d'attente des instructions *wait* est T'HIGH -T'NOW, où T'HIGH est la valeur maximale du type TIME, et T'NOW est la valeur courante du temps de simulation. Donc, si l'instruction *wait* n'a pas une condition ou une liste d'attente, le processus parent est suspendu indéfiniment.

En conclusion, pour notre sous-ensemble VHDL, les affirmations suivantes sont vraies :

- dans une unité de temps peuvent avoir lieu plusieurs cycles de simulation ;

- un cycle de simulation est exécuté dans au plus une unité de temps.

La simulation symbolique est un processus similaire à la simulation traditionnelle. La seule différence est que des objets de la description sont affectés avec des symboles au lieu des valeurs numériques. Ainsi, l'algorithme de simulation est le même.

Le processus de simulation nécessite un scénario de test, afin de fixer le comportement des entrées.

Soit le système d'équations récurrentes correspondant à une description VHDL :

$$C = \langle \mathcal{I}nput, \mathcal{L}ocals, \mathcal{O}ut, Init, \{E_x : x(t+1) = f_x(y(t), \ldots, z(t-1))\}_{x \in \mathcal{L}ocals \cup \mathcal{O}ut}, \mathcal{D}ecl \rangle$$

Un scénario Υ de test symbolique pour C est un ensemble d'équations, une pour chaque entrée du circuit. Chaque équation décrit le comportement de l'entrée dans le temps. Les équations peuvent être aussi des équations récurrentes.

$$\Upsilon = \{in : \mathbb{Z} \rightarrow \mathcal{T}\}$$

Par défaut, $in(t) = in_0$, $\forall t \in \mathbb{Z}$, où in_0 est un nouveau nom, qui est le symbole d'entrée pour in.

La composition d'un SER avec un scénario de test forme un système fermé (sans stimulus extérieur).

L'exécution symbolique d'un SER pour le cycle de simulation t_c est décrite par l'algorithme suivant :

Faire
$\forall x \in \mathcal{L}ocals \cup \mathcal{O}ut$ faire
Calculer $x(t_c + 1)$ en fonction de $x(t_c)$ à partir de $C \cup \Upsilon$;
Remplacer $x(t_c - 1)$ par $x(t_c)$;

Remplacer $x(t_c)$ par $x(t_c + 1)$;
Tant que Event$(S_{sync}(t_c)$

Tout d'abord, les entrées correspondant au cycle courant sont extraites à partir du scénario de test Υ. Les équations récurrentes sont exécutées pour une itération afin de calculer l'expression de $x(t_c + 1)$ en fonction de $x(t_c)$ et de $x(t_c - 1)$, pour toutes les memoires et les sorties x. Ce calcul correspond à un cycle de simulation. Ensuite, les valeurs précédentes et les valeurs courantes de x sont mises à jour. Ceci devrait correspondre à l'incrémentation du temps de simulation. Mais si la description n'est pas encore stable, alors le temps de simulation n'est pas incrémenté, car les cycles de simulation nécessaires pour la stabilisation se réalisent dans un temps 0 (ce sont des delta cycles). Ainsi, le temps de simulation courant reste t_c.

Le processus est répété jusqu'à ce que les valeurs se stabilisent : pour chaque objet x du circuit, la valeur future $x(t_c + 1)$ devient la valeur courante $x(t_c)$. Ceci correspond à un calcul du plus petit point fixe des fonctions qui décrivent les équations récurrentes. Du point de vue mathématique, l'existence d'un tel point fixe n'est pas garantie. Dans ce cas, nous considérons qu'il s'agit d'une erreur de conception, puisque cela correspond à un circuit dont les valeurs ne se stabilisent pas. Pendant la boucle de stabilisation, les entrées du circuit ne changent pas.

Un simulateur symbolique a été réalisé en Mathematica. Les détails de l'implémentation se trouvent dans [115]. Le simulateur utilise notre modèle basé sur des équations récurrentes et une partie des règles de réécriture présentées dans la section suivante.

2.8 Règles de réécriture pour la simplification d'expressions

Pendant l'exécution de l'algorithme de simulation, la taille des expressions peut devenir très grande. D'ailleurs, le problème de la complexité des expressions apparaît aussi pendant la génération des équations récurrentes, et pas seulement pendant la simulation symbolique. Afin de résoudre ce problème, nous avons défini plusieurs règles de simplification concernant les fonctions que nous avons introduites pendant la transformation du code VHDL.

La fonction IF

Nous rappelons la définition de la fonction IF :

IF(true, E, F)=E
IF(false, E, F)=F

La modélisation avec des fonctions IF a été utilisée par [91] afin d'implémenter les BDD dans le démonstrateur Nqthm. Dans ces travaux plusieurs théorèmes sont démontrés sur le comportement de la fonction :

Théorème 1 *La fonction IF(C,E,E) est équivalente à E.*

Théorème 2 *La fonction IF est distribuable sur d'autres fonctions f :*
f(E$_1$,..., IF(C,E,F), ...) est équivalente à IF(C, f(E$_1$, ..., E, ...) , f(E$_1$, ..., F, ...))

La preuve des théorèmes est immédiate en utilisant la définition de IF et la preuve par cas. A partir de sa définition et des théorèmes, nous déduisons des règles de simplification pour la fonction IF :

IF ($true$,E,F) $\rightarrow$ E
IF ($false$,E,F) $\rightarrow$ F
IF (C,E,E) $\rightarrow$ E
f (E$_1$, ..., IF(C,E,F) ... E$_n$) $\longrightarrow$ IF(C, f(E$_1$, ..., E ...) , f(E$_1$, ..., F ...)) $\forall f \neq IF$

On remarque que pour la dernière règle nous avons mis la restriction $f \neq IF$. Ceci est nécessaire si l'on veut que le système de réécriture converge. Si la restriction n'existe pas, et f est IF, alors la dernière règle peut s'appliquer indéfiniment.

Dans le même temps, les règles de transformation d'instructions séquentielles, introduisent de nombreuses fonctions IF imbriquées. Puisque nous voulons avoir des fonctions IF dans la forme la plus compacte, nous introduisons quelques règles de simplification applicables pour la distribution de la fonction IF sur elle-même.

IF (IF(C,E,F), G,H) $\longrightarrow$ IF(C, IF(E,G,H), IF(F,G,H))
IF (C$_1$, IF(C$_2$, E, F), F) $\rightarrow$ IF(C$_1$ and C$_2$, E, F)
IF (C$_1$, IF(C$_2$, E, F), E) $\rightarrow$ IF(C$_1$ and (not C$_2$), F, E)
IF (C$_1$, E, IF(C$_2$, E, F)) $\rightarrow$ IF(C$_1$ or C$_2$, E, F)
IF (C$_1$, F, IF(C$_2$, E, F)) $\rightarrow$ IF((not C$_1$) and C$_2$, E, F)

Théorème 3 *Les règles de simplification gardent la sémantique de la fonction IF.*

La démonstration du théorème est triviale en utilisant la preuve par cas.

Si les règles présentées auparavant ne peuvent pas s'appliquer, nous pouvons utiliser l'algorithme suivant :

simplify(True, False, IF (C, E, F)) =
si $C \in$ True alors *simplify*(True, False, E)
sinon si $C \in$ False alors *simplify*(True, False, F)
sinon
soit E^s=*simplify*(True$\cup${C}, False, E) et
F^s=*simplify*(True, False$\cup${C},F)
dans IF(C,E^s,F^s)

$simplify$(True, False, E) = E, si E n'est pas une fonction IF.

La fonction *simplify* prend en entrée un ensemble True d'expressions vraies, un ensemble False d'expressions fausses et une fonction IF, et elle renvoie une expression simplifiée. Pour une fonction IF, l'appel de la fonction à la forme $simplify(\emptyset, \emptyset,$ IF (C, E, F)).

L'algorithme utilise le fait que la condition C est vraie dans le branchement *then* et est fausse dans le branchement *else*. Si la condition est déjà dans l'ensemble des expressions vraies, alors l'expression E du branchement *then* continue à être simplifiée s'il s'agit d'une fonction IF, sinon l'algorithme s'arrête et l'expression est renvoyée. Si la condition est dans l'ensemble des expression fausses, le même processus s'applique à l'expression du branchement *else*. Si l'on ne peut rien dire sur la condition, alors on suppose d'abord qu'elle est vraie et on simplifie le branchement *then*, et ensuite on suppose qu'elle est fausse et on simplifie le branchement *else*. Finalement la fonction IF avec ses arguments simplifiés est retournée.

Les fonctions *nth* **et** *replace*

Les fonctions *nth* et *replace* sont duales donc nous pouvons décrire des règles de simplification pour leur composition :

a) L'élément qui se trouve sur la position *nil* d'un objet est l'objet lui-même.

$nth(nil, x) \rightarrow x$

b) Si la position *nil* est modifiée, l'objet ne change pas.

$replace([(nil, F)], x) \rightarrow F$

c) L'élément sur la position p d'un objet après qu'un remplacement a eu lieu, a la valeur définie par le remplacement si la position a été modifiée.

$nth(p,\ replace([(\mathrm{p},\ \mathrm{F}),\mathrm{L}],\ x)) \longrightarrow \mathrm{F}$

Exemple :

Soit le code VHDL :

```
a(1,2) := F ;
b := 2 + a(1,2) ;
```

Conformément à nos règles de transformation VHDL, le code dévient :

$a := replace([((1,2),F)],a) \parallel$
$b := 2 + nth((1,2), replace([((1,2),F)],a))$

Maintenant la règle c) est applicable et le résultat final est :

$a := replace([((1,2),F)],a) \parallel$
$b := 2 + F.$

d) Si l'élément sur la position p n'a pas été modifié par le remplacement, sa valeur est identique à la valeur avant le remplacement.

$nth(p,\ replace([\text{q, F,L}],\ x)) \longrightarrow nth(p,\ replace(\text{L},\ x))$, si p et q sont complètement disjointes (ne désignent pas les mêmes éléments).

Nous considérons que deux positions sont complètement disjointes si : soit les positions sont des listes de la même longueur et elles ont au moins un élément différent ; soit les positions sont de longueur différente et elles ne sont pas des sous-listes l'une de l'autre.

Par exemple, soit le type composé `matrice` et la variable `m` de type `matrice`.

```
type word is array (0 to 31) of BIT;
type line is array (0 to 256) of word;
type matrice is array (0 to 256) of line;
```

Les positions (1,1,2) et (1,1) de `m` ne sont pas disjointes, car l'élément à la position (1,1) inclut l'élément à la position (1,1,2). Par contre, les positions (1,1,2) et (2) sont disjointes. Idem pour (1,1,2) et (1,1,3).

Formellement, p et q sont disjointes si est seulement si $p \neq nil$ et $p \neq nil$ (donc soit $p = (a, P)$ et $q = (b, Q)$) et soit $a \neq b$, soit $a = b$ et P et Q sont disjointes.

Maintenant soit le code VHDL :

```
a(2,2) := E;
b := 2 + a(1,2);
```

Conformément aux règles de transformation présentées dans les sections antérieures, le code dévient :

$a := replace([((2,2),E)],a) \parallel$
$b := 2 + nth((1,2), replace([((2,2),E)],a))$

Maintenant la règle d) est applicable. Le résultat final est :

$a := replace([((2,2),E)],a) \parallel$
$b := 2 + nth((1,2),a)$

e) Les remplacements pour un objet sont cumulables.

$replace(\mathrm{L}_1, replace(\mathrm{L}_2, x)) = replace(integrate(\mathrm{L}_1, \mathrm{L}_2), x)$

L'opérateur *integrate* calcule la liste de remplacements après des affectations séquentielles du même objet. Si une position est réécrite par le deuxième remplacement, seule cette écriture est observable.

$integrate(nil, \mathrm{L}_2) = \mathrm{L}_2$
$integrate(\mathrm{L}_1, \mathrm{L}_2) = \mathrm{L}_1$ & { (p,F) | $(p,\mathrm{F}) \in \mathrm{L}_2$ et $\forall (q,E) \in \mathrm{L}_1$, p et q ne sont pas identiques }

Exemple 1 :

Soit le code VHDL :

```
a(2,2) := E;
a(1,2) := F;
```

Après transformation, le code devient :

$a := replace([((1,2),F)], replace([((2,2),E)],a))$.

Le résultat de la simplification avec la règle e) est :

$a := replace([((1,2),F),((2,2),E)],a)$.

Exemple 2 :

Soit le code VHDL :

```
c(2,2) := E;
c(2,2) := F;
```

L'affectation parallèle correspondant à ce code VHDL est :

$c := replace([((2,2),F)], replace([((2,2),E)], c))$.

Après l'application de la règle e), l'affectation devient :

$c := replace([((2,2),F)], c)$.

Théorème 4 *Les règles de simplification pour nth et replace gardent la sémantique des fonctions (sont correctes).*

Démonstration

Les détails de la preuve se trouvent dans l'annexe 7.

Les deux premières règles, a) et b), sont déduites de la définition des fonctions.

Pour la troisième règle, c), il revient à prouver que $nth(p, replace([(p,F),L],x)) = F$. Nous allons utiliser la définition de nth.

Pour démontrer la règle d), il faut prouver que, si p et q sont complètement disjointes, alors $nth(p, replace([(q,F),L],x))$ est égal à $nth(p, replace(L,x))$.

Pour démontrer la règle e), il faut montrer que

$replace(L_1, replace(L_2, x)) = replace(integrate(L_1, L_2), x)$.

Nous allons utiliser la récurrence sur la longueur de la liste des positions L_1.

QED

2.9 Un modèle pour les circuits synchrones

Les circuits synchronisés par horloge ont un comportement très régulier. Les modifications des valeurs d'objets du circuit ont lieu sous la condition qu'un événement a été déclenché pour l'horloge. L'événement sur l'horloge est appelé front. Il peut être descendant, si la valeur de l'horloge a changé de 1 à 0, ou montant si le changement est inverse, de 0 à 1.

Il existe deux manières de décrire des circuits synchronisés par horloges : l'affectation d'objets a lieu seulement sur un front (soit descendant, soit montant) ou l'affectation d'objets a lieu sur les deux fronts.

Dans les deux cas, nous obtenons un modèle basé sur les équations récurrentes en utilisant la simulation symbolique. Généralement l'horloge a comme première valeur 0, pour la phase d'initialisation, et une période est considérée comme la composition de deux cycles temporels. Pour le premier cycle l'horloge est à 1, et pour le deuxième l'horloge est à 0. Pour obtenir le comportement du circuit pendant un cycle d'horloge, il suffit de simuler symboliquement le SER pour les deux cycles temporels. Ainsi, le résultat de la simulation symbolique est toujours un SER.

La simulation symbolique peut être utilisée de différentes manières afin d'extraire le comportement du circuit :

- à part l'horloge, d'autres signaux d'entrée peuvent avoir des valeurs numériques pendant la simulation. Par exemple, dans le cadre d'un projet nous avons reçu un circuit

qui implémente deux algorithmes de hachage : SHA-1 et SHA-256. Le circuit a donc deux modes de fonctionnement (il réalise SHA-1 ou SHA-256), définis par l'entrée booléenne ch_mod. Pour faciliter la tâche de vérification, nous avons choisi d'extraire un modèle du circuit en mode SHA-1 et un deuxième modèle en mode SHA-256. Ainsi, dans un premier temps, nous avons simulé symboliquement le circuit pour un cycle d'horloge avec ch_mod à 1. Nous avons montré que le modèle obtenu, un SER, implémente correctement l'algorithme SHA-1. Ensuite, le circuit a été simulé pour un cycle d'horloge avec ch_mod à 0, et prouvé par la suite en conformité avec l'algorithme SHA-256.

- les circuits avec un comportement régulier représentent un autre cas particulier auquel des schémas de simulation peuvent s'appliquer. Si par exemple le circuit fournit une réponse significative tous les trois cycles d'horloge, alors il est plus intéressant de construire un SER sur les trois cycles, que de simuler la description seulement pour un cycle.
- les deux premières façons de simuler les circuits peuvent se combiner, afin d'obtenir le modèle le plus compact concernant l'information nécessaire pour la preuve.

Conclusion

Dans ce chapitre, nous avons présenté le modèle d'équations récurrentes et comment l'obtenir à partir d'une description en VHDL. Le sous-ensemble de VHDL pris en compte inclut les objets de type composé, les boucles, les sous-programmes, les processus avec plusieurs instructions de synchronisation et les composants.

Dans un premier temps, le modèle SER extrait à partir d'un circuit est au niveau du cycle de simulation. Nous avons montré comment utiliser la simulation symbolique afin d'obtenir un modèle SER au niveau du cycle d'horloge, ou pour des comportement particuliers du circuit.

Dans le chapitre suivant, nous présentons le démonstrateur ACL2 et comment traduire un SER dans sa logique.

Chapitre 3

Modélisation d'un circuit VHDL dans ACL2

Ce chapitre est structuré en deux parties. Dans la première partie, nous présentons brièvement le système ACL2 afin de faciliter la lecture de la suite du document. Dans la deuxième partie, nous montrons comment un circuit représenté par un système d'équations récurrentes est modélisé dans ACL2.

3.1 Introduction au démonstrateur de théorèmes ACL2

ACL2[1] [72, 73] est un système qui contient :
- un langage de programmation (un sous-ensemble de Lisp, donc la notation est préfixée)
- une logique de premier ordre avec arithmétique sur des fonctions totales récursives et des objets construits par récurrence.
- un démonstrateur de théorèmes.

L'interface du système avec l'utilisateur est un "shell", où sont soumis des événements. Les événements les plus courants sont la définition d'une fonction (`defun`), la définition d'un théorème (`defthm`) ou l'évaluation d'un appel de fonction (`(cons 2 3)`). Chaque soumission d'événement génère une réaction de la part du système : soit le résultat de l'évaluation si c'est le cas d'un appel de fonction, soit des informations sur l'acceptation ou non de l'événement dans les autres cas.

La sortie du démonstrateur est très verbeuse et informelle, donnant des détails sur les heuristiques utilisées et les décisions prises, ce qui est très utile pour contrôler et comprendre le comportement de l'outil. Donc, si un théorème est prouvé dans ACL2, la sortie du démonstrateur ne fournit pas une preuve formelle, par contre le fait que le théorème a été accepté, garantit qu'il existe une telle preuve. Dans le même temps, puisque la logique d'ACL2 n'est pas décidable, si le démonstrateur ne réussit pas à prouver un théorème, on ne peut rien déduire sur la valeur de vérité de la formule à prouver. Mais dans ce cas, l'inspection des informations données en sortie par le démonstrateur aide à trancher : soit le but $false$ a été obtenu donc une erreur a été mise en évidence, soit la preuve a besoin de théorèmes intermédiaires ou de stratégies.

[1] A Computational Logic for Applicative Common Lisp

Par la suite, nous détaillons les aspects spécification et preuve du démonstrateur, en insistant sur les mécanismes que nous avons fréquemment utilisés.

3.1.1 Comment spécifier en ACL2

Le langage de programmation d'ACL2 utilise un sous-ensemble de Common Lisp : les fonctions, en ignorant les parties qui produisent des effets de bord comme les variables globales et la destruction de données.

Classes d'objets et expressions

Les objets de base d'ACL2 appartiennent à une des catégories suivantes :
- nombres entiers : 345, -345, rationnels : 45/78, complexes : #c(18 67) ;
- caractères : #d ;
- chaînes de caractères : "Vous lisez une thèse" ;
- symboles : nil, sha : : stop ;
- paire pointée (appelés aussi *cons*) : `(d . nil)` , `((1 . m) (3 . n))`

Les nombres sont généralement écrits dans la base 10, mais les notations binaires, octales et hexadécimales sont acceptées. Par exemple, 18 peut être écrit #b10010, #o21 ou #x12. Les symboles les plus fréquents sont *t* et *nil* et représentent les valeurs vrai et faux dans la logique d'ACL2. Le symbole *nil* représente aussi la liste vide ().

Les paires pointées sont les objets les plus utilisés dans ACL2. Elles se définissent à l'aide de l'opérateur '.'. Soient a et b deux symboles, `(a . b)` est une paire pointée. Les listes sont des paires particulières, où l'élément de la partie droite est soit *nil* soit une liste. Pour les listes, le point et les parenthèses peuvent être éliminés. Par exemple `(3 . nil)` et `(3)` sont équivalentes. Aussi, `(a . (b .(c . nil)))` , `(a .(b .(c)))`, `(a . (b c))`, `(a b c)` sont équivalentes.

Contrairement aux autres langages de programmation qui ont des instructions, procédures, modules, etc., un programme ACL2 est composé seulement d'expressions.

Une expression est soit un *symbole de variable* comme x, y, etc., soit un *symbole constant* comme *t*, *nil*. L'utilisateur peut aussi introduire des symboles constants avec la commande `defconst`. Par exemple `(defconst *x* 10)` définit le symbole `*x*` qui est évalué à 10.

Un autre type d'expression est *l'expression constante*. Les nombres, les caractères et les chaînes de caractères entrent dans cette catégorie. Une expression constante est aussi un objet ACL2 précédé de '. La valeur d'une telle expression est l'objet ACL2. Par exemple la valeur de `'car` est le symbole `car`. Ce type d'expression est beaucoup utilisé dans la définition des macros, que nous présentons plus tard.

Le dernier type d'expression est *l'expression fonctionnelle* $(f\ v_1 \ldots v_n)$ qui correspond à l'application d'un symbole fonctionnel f de n variables, à n expressions v_1, ..., v_n.

`(cons` x y`)`	construit un couple $\langle x, y \rangle$
`(car` x`)`	le premier élément de x, si x est un couple ; `nil` autrement
`(cdr` x`)`	le deuxième élément de x, si x est un couple ; `nil` autrement
`(consp` x`)`	`t` si x est un couple ; `nil` autrement
`(if` x y z`)`	z si x est `nil` ; y autrement
`(equal` x y`)`	`t` si x est y ; `nil` autrement

FIG. 3.1 – Des fonctions primitives d'ACL2

Fonctions

Une *fonction* f est définie avec la commande `(defun` f $(v_1 \ldots v_n)$ β`)`, où v_i sont les arguments de la fonction, et β est le corps de la fonction. Les fonctions d'ACL2 sont totales, étant définies sur tous les domaines.

La figure 3.1 énumère quelques fonctions primitives d'ACL2.

La définition des fonctions en ACL2 est soumise à plusieurs règles :

- le corps de la fonction ne contient pas d'autres variables que v_i (les variables globales ne sont pas acceptées).
- toute fonction utilisée dans le corps de la fonction, β, doit être définie auparavant.

La plupart des fonctions définies en ACL2 sont *récursives*, car ACL2 n'a pas de structures de contrôle itératives ou de fonctions d'ordre supérieur. La définition récursive est soumise à une règle de plus : ACL2 doit prouver qu'il existe un argument (ou une combinaison des arguments) de la fonction dont la mesure décroît selon une relation bien fondée, sous les conditions qui génèrent la récursivité. Ceci est connu sous le nom de *Principe de définition*.

Une relation $\preceq_R$ est bien fondée s'il n'existe pas de séquence infinie $x_1, x_2, x_3, \ldots$ telle que : $x_{i+1} \preceq_R x_i$, $\forall 1 \leq i$. La validité de la mesure assure la terminaison de la récursivité. Dans la plupart des cas, ACL2 est capable de trouver automatiquement la mesure d'une fonction récursive.

Nous définissons la fonction récursive `fib` qui calcule l'élément n de la série de Fibonacci. `zp` est un prédicat qui reconnaît les objets qui ne sont pas des entiers positifs.

```
(defun fib (n)
  (if (zp n) 0
    (if (equal n 1) 1
      (+ (fib (- n 1)) (fib (- n 2))))))
```

Le démonstrateur trouve que n est l'argument de la fonction sur lequel appliquer la mesure. La fonction fib est acceptée lorsqu'il est prouvé que la mesure décroît. Les sorties complètes du démonstrateur pour les exemples de cette section se trouvent dans l'annexe 8. Il existe des cas où le démonstrateur n'est pas capable de déduire la mesure de la fonction. Prenons l'exemple de la fonction `gcdD(a b)` qui calcule le plus petit diviseur commun de

deux entiers positifs conformément à l'algorithme de Dijkstra :

```
(defun gcdD (a b)
  (cond ((zp a) b)
        ((zp b) a)
        ((equal a b) a)
        (t  (if (< a b)
                (gcdD a (- b a))
              (gcdD (- a b) b)))))
```

ACL2 ne trouve pas de mesure, et donc la fonction n'est pas acceptée. La fonction `gcdD` est pourtant correcte, car la somme des deux arguments décroît, et donc, la fonction se termine.

Si le démonstrateur ne trouve pas la bonne mesure, l'utilisateur a la possibilité de l'indiquer dans la partie *déclarations* de la définition. Lorsque la mesure est précisée, le démonstrateur accepte la définition après avoir prouvé que la mesure est correcte.

```
(defun gcdD (a b)
 (declare (xargs :measure (acl2-count (+ a b))))
   (cond ((zp a) b)
         ((zp b) a)
         ((equal a b) a)
         (t  (if (< a b)
                 (gcdD a (- b a))
               (gcdD (- a b) b)))))
```

Les déclarations sont des indications pour le compilateur Lisp ou ACL2. Les plus fréquemment utilisées sont les formes :

- `(declare (ignore` v_1 `...))` indique que le paramètre v_1 n'est pas utilisé dans le corps de la fonction.
- `(declare (xargs :guard` g`))` indique que la garde d'une fonction est g. L'utilisation des gardes est étroitement liée à l'exécution efficace des fonctions. Une discussion à ce sujet est présentée à la fin de la section.
- `(declare (xargs :measure` m`))` indique la mesure à utiliser pour trouver la terminaison d'une fonction. m est une expression qui est évaluée à un ordinal ACL2 et décroît avec l'appel récursif. La mesure est le plus souvent calculée à l'aide de la fonction `acl2-count`, une primitive d'ACL2.

La récursivité croisée est définie en ACL2 avec la commande :

```
(mutual-recursion
   (defun  f1 ...)
     ...
   (defun  fn ...))
```

Macros

Les macros sont des fonctions qui s'appliquent au niveau syntaxique. Une fonction définie avec `defun` s'applique à des objets pour obtenir des valeurs. Une macro s'applique à des objets pour obtenir des expressions.

La définition d'une macro a la forme `(defmacro` m *parametres* β`)`.

Voici une définition de macro en ACL2 :

```
(defmacro cadr (x)
   (cons 'car (cons 'cdr (cons x nil))))
```

La macro crée une liste qui a comme premier élément le symbole `car`, comme deuxième élément le symbole `cdr` et le reste de la liste est l'objet x. L'application de la macro `cadr` à l'expression `(cons x a)` retourne l'expression `(car (cdr (cons x a)))`. L'évaluation de la fonction `cadr` revient à évaluer l'expression que la macro construit. En effet, la fonction `cadr` extrait le deuxième élément d'une liste.

Généralement, les macros sont utilisées pour définir des stratégies de preuve ou de programmation. Par exemple, la macro `def-func` définit une fonction qui, appliquée à une liste, crée une nouvelle fonction ayant le premier élément de la liste comme nom et le deuxième élément comme corps. `read-sym` est une fonction définie par l'utilisateur qui extrait les variables symboles de la liste `l`.

```
(defmacro def-func (l)
   (list 'defun (car l) (read-sym (cadr l)) (cadr l)))
```

Si la macro est appliquée à la liste `(sum (+ y x))`, le résultat est la fonction :

```
(defun sum (y x) (+ y x))
```

Donc l'évaluation par ACL2 de l'appel `(def-func (sum (+ y x))` revient à soumettre au démonstrateur la définition de `sum`. Voici la sortie du démonstrateur.

```
 ACL2 !>(def-func (sum (+ y x)))
Since SUM is non-recursive, its admission is trivial.  We observe that
the type of SUM is described by the theorem (ACL2-NUMBERP (SUM Y X)).
We used primitive type reasoning.
Summary
Form :  ( DEFUN SUM ...)
Rules : (( :FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings :  None
Time :  0.00 seconds (prove : 0.00, print : 0.00, other : 0.00)
 SUM
```

Une autre utilisation des macros est dans le cas des fonctions avec un nombre variable de paramètres. Soit par exemple la fonction `sum` définie auparavant. Initialement, elle a

été définie pour deux paramètres, mais d'habitude, l'addition est appliquée avec plusieurs paramètres. Donc, nous souhaitons que la fonction `sum` ait la même caractéristique. Techniquement, ceci est réalisé par une macro qui transforme l'appel avec plusieurs paramètres en des appels répétés de la fonction binaire. Par exemple, `(sum a b c d)` est transformé en `(sum a (sum b (sum c d)))`.

Exécution efficace en ACL2

Un des points forts d'ACL2 par rapport aux autres démonstrateurs est son exécutabilité. Un programme est à la fois une spécification dans la logique du démonstrateur et un modèle exécutable. Ainsi, l'exploration du modèle et sa validation par des jeux de test est possible. En plus, ACL2 repose sur le compilateur Lisp, dont la vitesse d'exécution est comparable avec celle de C/C++ et nettement mieux que celle de JAVA [46].

Malheureusement, le démonstrateur ne bénéficie pas toujours de la vitesse de calcul de Lisp, car les fonctions primitives ne sont pas toujours équivalentes avec leurs correspondants en Lisp. En effet, les fonctions ACL2 sont des extensions des fonctions Lisp. Ceci est une conséquence de la logique qui demande que toute fonction soit *totale*, c'est à dire qu'elle peut avoir tout type d'argument. En Common Lisp les fonctions peuvent être partielles. Même si le langage n'est pas typé, chaque primitive a un domaine de définition et le comportement de la primitive n'est pas défini en dehors du domaine. Par exemple, le comportement de la fonction + appliquée aux arguments autres que numériques, n'est pas spécifié dans le manuel de référence de Lisp : `(+ nil 3)` n'est pas défini et une erreur est signalée dans la plupart des implémentations.

En ACL2, les fonctions primitives de Common Lisp sont étendues par des axiomes pour les arguments qui sont en dehors du domaine de définition. D'habitude, l'extension est réalisée par la conversion des 'mauvais' arguments vers des 'bons'. Si on reprend l'exemple de la fonction +, tout argument qui n'est pas un nombre est transformé en 0. Mais dans le même temps, les domaines de définition de Lisp sont mémorisés comme des gardes. Ainsi, ACL2 a deux comportements possibles : avec vérification des gardes ou sans. Par défaut la vérification des gardes est activée.

Quand la vérification des gardes est active, et les arguments d'une fonction sont en dehors de son domaine de définition, une erreur est signalée. Par exemple si l'expression `(+ nil 3)` doit être évaluée, ACL2 retourne le message :

```
ACL2 !>(+ nil 3)
ACL2 Error in TOP-LEVEL :  The guard for the function symbol BINARY-
+, which is (AND (ACL2-NUMBERP X) (ACL2-NUMBERP Y)), is violated by
the arguments in the call (+ NIL 3).
```

Si par contre, la vérification des gardes est inactivée, ACL2 utilise ses axiomes pour l'évaluation de l'expression, et donc `(+ nil 3)` est évalué à 3.

La vérification de gardes pour une fonction implique la preuve que la fonction respecte

les gardes de toutes les fonctions appelées dans son corps. Donc pour tout paramètre d'entrée, les fonctions ont des arguments dans leur domaine de définition. Si une fonction a les gardes vérifiées, elle est dite *compatible Lisp*, et son exécution est d'habitude plus rapide, puisqu'il n'y a plus de vérification de type. Si la vérification des gardes n'a pas lieu, alors la fonction est évaluée conformément aux axiomes ACL2.

Dans la définition d'une fonction, l'utilisateur peut spécifier une garde dans les déclarations. Par exemple, la fonction `gcdD` définie auparavant, s'applique seulement aux entiers positifs. Dans ACL2 elle est définie pour tous les domaines. En ajoutant la déclaration des gardes, pour que la fonction soit acceptée, deux preuves sont réalisées : une pour la terminaison et une pour la vérification des gardes. Une fois acceptée, la fonction est considérée compatible avec Lisp, et donc son temps d'exécution sera plus court.

```
(defun gcdD-gardes (a b)
 (declare (xargs :measure (acl2-count (+ a b))
                 :guard (and (integerp a) (<= 0 a)
                             (integerp b) (<= 0 b))))
   (cond ((zp a) b)
         ((zp b) a)
         ((equal a b) a)
         (t  (if (< a b)
                 (gcdD-gardes a (- b a))
               (gcdD-gardes (- a b) b)))))
```

La manière dont les fonctions récursives sont définies influe aussi sur leur temps d'exécution. Il existe deux types de récursivité : la récursivité terminale, et la récursivité non-terminale. Dans le cas de la récursivité non-terminale, la fonction est appelée avec les nouveaux paramètres, et une fois terminée, son résultat va remonter toutes les feuilles de calcul. La récursivité dans la définition de la fonction qui calcule la suite de Fibonacci, `fib`, est non-terminale.

Dans le cas de la récursivité terminale, la fonction a un accumulateur qui contient le résultat de l'appel récursif, et évite ainsi de tout désempiler lors du retour. Par exemple, la définition de `gcdD` est de type terminal.

Les fonctions de type terminal sont plus efficaces, justement parce qu'elles évitent l'empilement des appels dans la mémoire.

3.1.2 Comment prouver en ACL2

La logique ACL2

ACL2 est une logique du premier ordre avec arithmétique (une description détaillée de la logique du démonstrateur se trouve dans [74]). Voici la définition des connecteurs logiques d'ACL2 (`iff` dénote l'opérateur d'équivalence) :

1. `t` $\neq$ `nil`
2. `x` $\neq$ `nil` $\rightarrow$ `(if x y z)` = `y`
3. `x` = `nil` $\rightarrow$ `(if x y z)` = `z`
4. `(equal x y)` = `nil` $\vee$ `(equal x y)` = `t`
5. `x` = `y` $\leftrightarrow$ `(equal x y)` = `t`
6. `(consp x)` = `nil` $\vee$ `(consp x)` = `t`
7. `(consp (cons x y))` = `t`
8. `(consp nil)` = `nil`
9. `(car (cons x y))` = `x`
10. `(cdr (cons x y))` = `y`
11. `(consp x)` = `t` $\rightarrow$ `(cons (car x) (cdr x))` = `x`

FIG. 3.2 – Des axiomes pour les fonctions primitives d'ACL2

```
(defun not (p) (if p nil t))
(defun and (p q) (if p q nil))
(defun or (p q) (if p p q))
(defun implies (p q) (if p (if q t nil) t))
(defun iff (p q) (and (implies p q) (implies q p)))
```

L'égalité entre deux expressions est une formule atomique de la logique. Les formules sont construites de manière récursive, à partir des formules atomiques, en utilisant les opérateurs logiques. ACL2 implémente les axiomes et les règles de déduction de la logique classique [121].

Les fonctions primitives d'ACL2 sont aussi définies par des axiomes. La figure 3.2 énumère quelques axiomes correspondant aux fonctions présentées dans la figure 3.1.

Dans le même temps, chaque fois qu'une nouvelle définition `(defun` f `(`$v_1 \ldots v_n$`)` β`)` est acceptée par le démonstrateur, l'axiome `(`f $v_1 \ldots v_n$`)` `=` β est ajouté au système.

Une propriété à prouver est définie avec la commande `(defthm t` β`)`. Après acceptation par ACL2, elle est transformée en règle de déduction.

Le *Principe de Récurrence* permet de démontrer des propriétés faisant appel à des fonctions récursives. Pour prouver une propriété :

- On montre que la propriété est vraie pour le (ou les) cas de base.

- Si X est la variable de récurrence, on suppose que la propriété est vraie pour les éléments plus petits que X (au sens de la mesure), et on montre qu'elle est encore vraie pour X.

C'est en grande partie grâce au principe de définition qu'ACL2 est capable de construire automatiquement le schéma de récurrence.

Les deux principes (de récurrence et de définition) sont basés sur la notion d'ordre bien fondé, qui, dans le cas d'ACL2, est l'ordre sur les ordinaux. Ainsi, tout objet ACL2 a une

dimension, mesurée par la fonction `acl2-count`.

Un autre principe de la logique est le *Principe d'Extension* qui est réalisé par la commande `encapsulate` [76]. L'encapsulation introduit sous certaines contraintes des symboles de fonctions non définis. Par exemple, la commande suivante introduit une contrainte ϕ sur le symbole f (sous la forme du théorème `thm-1`).

```
(encapsulate (((f x1 ... xn) => *))
    (local (defun f (x1 ... xn) β))
    (defthm thm-1 φ))
```

L'encapsulation est acceptée si l'événement `defun` est accepté et `thm-1` est un théorème. Dans l'encapsulation, l'événement `defun` est précédé du mot `local`. Ceci signifie que la théorie courante est étendue provisoirement avec la définition de f, afin d'évaluer l'acceptation de la contrainte ϕ. A la sortie de la commande `encapsulate`, la définition de f est éliminée de la théorie. Dans le même temps, la théorie est étendue avec l'axiome : f est contrainte avec ϕ. Ainsi, la fonction f n'est pas définie, mais on sait qu'elle a la propriété ϕ. Pour toutes les fonctions encapsulées, il est obligatoire de fournir une fonction locale. Le démonstrateur s'assure ainsi qu'il existe au moins une fonction qui a les propriétés énoncées. La fonction locale fournie est appelée témoin local.

Mécanique de preuve

Un théorème soumis au démonstrateur est généralement décomposé en lemmes, et chaque lemme doit être prouvé par ACL2. Le système utilise plusieurs techniques de preuve dans l'ordre suivant :

- simplification : expansion des définitions et applications de procédures de décision, réécriture, normalisation, etc. ;
- élimination des destructeurs : cette technique s'applique pour les objets définis par récurrence ;
- utilisation d'équivalences : prise en compte d'hypothèses qui représentent une égalité, comme l'hypothèse de récurrence ;
- généralisation pour obtenir une propriété plus forte, qui souvent est plus facile à prouver ;
- élimination des termes non pertinents pour la formule à prouver ;
- récurrence.

Chaque technique applicable transforme la formule à prouver dans d'autres formules à prouver, et le processus recommence avec la simplification. Si une technique n'est pas applicable, la technique suivante prend le contrôle de la preuve.

Si aucune des techniques ne réussit à réduire la formule à vrai, alors la preuve s'arrête ou, dans certains cas, elle boucle. On ne peut donc rien conclure sur la correction de la propriété, puisque la logique d'ACL2 est indécidable.

Un des premiers pas dans l'étape de simplification est l'expansion des fonctions non récursives. Donc si le système contient de telles définitions, il est indiqué de créer une règle de réécriture pour la fonction et désactiver sa définition par la suite. Sinon, il se peut que des règles de déduction pour la fonction ne s'appliquent jamais. Par exemple, soit la fonction non récursive *natp* qui est un prédicat qui reconnaît un nombre naturel. La spécification de *natp* est :

$$natp(x) = \begin{cases} true \text{ si } x \in \mathbb{N} \\ false,\ autrement \end{cases}$$

Voici le code ACL2 de *natp* :

```
(defun natp (x)
 (and (integerp x) (<= 0  x)))
```

Maintenant, soit le théorème *natp-prod* qui décrit la propriété que le produit de deux éléments naturels est un naturel : $\forall x, y \in \mathbb{N}.x \times y \in \mathbb{N}$. Voici le code ACL2 pour *natp-prod* :

```
(defthm natp-prod
  (implies (and (natp x) (natp y))
      (natp (*  x y))))
```

Le théorème `natp-prod` ne va jamais s'appliquer puisque à chaque fois que le terme `(natp x)` est rencontré, celui-ci est remplacé par son corps `(and (integerp x) (<= 0 x))`, et donc `(natp x)` disparaît de la propriété à prouver. Pour cela il est indiqué de désactiver la définition de *natp*, i.e. interdire son expansion.

La commande `(in-theory (disable` *nom*`))` permet d'éliminer la règle *nom* des règles à appliquer. Ainsi, `(in-theory (disable natp))` désactive la définition de *natp* et toutes les règles pour *natp* sont applicables. La commande est réversible : `(in-theory (enable` *nom*`))` ajoute la règle *nom* à l'ensemble des règles à appliquer.

ACL2 n'est pas un démonstrateur interactif car une fois la preuve démarrée, l'utilisateur ne peut pas agir sur son déroulement (sauf bien sûr pour l'arrêter). Par contre, il a la possibilité de donner des instructions pour la stratégie de preuve, lors de la définition du théorème, par le mot clé `hints`. Les instructions plus fréquemment utilisées sont :

- `:induct` *induction-scheme* pour suggérer un schéma de récurrence.
- `:use` *lemma-instance* pour utiliser l'instance d'une propriété déjà prouvée.
- `:do-not` *list-processes* pour ne pas appliquer les techniques énumérées dans la liste.

Par défaut, un théorème est considéré comme une règle de réécriture. L'utilisateur a la possibilité de le spécifier s'il veut que le théorème appartienne à une autre catégorie de règles, par le mot clé `:rule-classes`. Les catégories les plus utilisées, sauf `rewrite` sont :

- `nil` si l'on veut que le théorème ne soit pas utilisé par le démonstrateur dans la déduction d'autres théorèmes.

`(defconst *`sym`* 'const)`	définit la constante *sym*
`(defun` f `(`v_1 `...` v_2`)` $corps$`)`	définit la fonction f
`(defmacro` m $args$ $corps$`)`	définit le macro m
`(mutual-recursion`	définit n fonctions
`(defun` f_1 `...)`	mutuellement récurrentes
`...`	
`(defun` f_n `...))`	
`(defthm` th $prop$`)`	définit le théorème th
`(encapsulate`	définit l'encapsulation
`(`$s_1 \ldots s_n$`)`	de n symboles de fonction ayant
`(`$ev_1 \ldots ev_m$`))`	les signatures $s_1 \ldots s_n$
`(include-book` nom`)`	inclut la bibliothèque nom
`(in-theory (disable` r`))`	élimine la règle r de la théorie courante
`(in-theory (enable` r`))`	introduit la règle r dans la théorie courante

FIG. 3.3 – Des événements ACL2

- `forward-chaining` : une règle de ce type enrichit le contexte du théorème mais sans réécrire le but.
- `type-prescription` : pour la déduction de types. La déduction utilise seulement les règles de la catégorie `type-prescription`. Chaque fois qu'une définition est acceptée par le démonstrateur, une règle de déduction de type pour la définition est générée.

Des événements acceptés par le démonstrateur peuvent être réutilisés dans d'autres sessions de preuve ou dans d'autres projets. ACL2 a un système de bibliothèques ('book' dans le langage du démonstrateur) et la commande `(include-book` nom`)` ajoute au projet en cours l'ensemble de définitions et théorèmes de la bibliothèque nom.

La Figure 3.3 résume les événements d'ACL2 introduits dans cette section.

3.2 Traduction d'un circuit vers ACL2 en utilisant les SER

Soit P un circuit décrit en VHDL. Dans un premier temps, nous extrayons le système d'équations récurrentes, conformément à la méthode exposée dans le chapitre 3. Le SER résultant est simulé symboliquement afin d'obtenir un modèle stable qui représente le comportement du circuit pour une période fixe de temps. Nous allons montrer par la suite comment créer à partir du SER un modèle récursif en ACL2.

Soit $SER_P = \langle \mathcal{I}n, \mathcal{L}ocals, \mathcal{O}ut, Init, \{E_x : x(t+1) = f_x(y(t), \ldots)\}_{x \in \mathcal{L}ocals \cup \mathcal{O}ut}, Decl \rangle$, le système correspondant à la description P, où $Decl = (T_I, T_L, T_O, Types, Fonctions)$.

Nous différencions plusieurs catégories à modéliser en ACL2 : les déclarations de type ($Types$), de fonction ($Fonctions$), et d'objets (T_I, T_L et T_O), les fonctions d'équations récurrentes et le système dans son intégralité. Par la suite, nous supposons qu'il n'existe pas de conflit de nom avec des mots clé d'ACL2 pour la description VHDL, et donc pour le SER.

$Member(x, L)$	`(member x L)`	vérifie si x est un élément de L
$Consp(L)$	`(consp L)`	vérifie si L est une paire
$First(L)$	`(car L)`	le premier élément de la liste L
$Firstn(n, L)$	`(firstn n L)`	les n premiers éléments de la liste L
$Rest(L)$	`(cdr L)`	le reste de la liste L, après l'élimination du premier élément
$Restn(n, L)$	`(nthcdr n L)`	le reste de la liste L, après éliminer les n premiers éléments
$Empty(L)$	`(endp L)`	vérifie si la liste L est vide
$Len(L)$	`(len L)`	la longueur de la liste L
$Nth(i, L)$	`(nth i L)`	l'élément i de la liste L
$Update\text{-}Nth(i, val, L)$	`(update-nth i val L)`	remplace la valeur de l'élément i par val dans L
$Listp(L)$	`(true-listp L)`	vérifie si la liste est bien construite
$Append(L_1, \ldots, L_n)$	`(append` L_1 ... L_n`)`	la concaténation d'arguments

FIG. 3.4 – Des opérations sur des listes

Afin de faciliter la lecture des sections suivantes, nous résumons dans la Figure 3.4 quelques opérations sur les listes, en donnant leurs spécifications et leurs correspondants ACL2.

3.2.1 Les déclarations de type

ACL2 n'est pas un système typé, mais comme nous l'avons vu, il est important pour la preuve et l'exécution d'avoir des prédicats qui reconnaissent un type. Les types du SER sont les mêmes que les types VHDL déclarés dans la description initiale. Ainsi, pour chaque type de VHDL, nous souhaitons avoir un prédicat en ACL2. Certains types prédéfinis de VHDL, comme INTEGER, NATURAL, REAL, BOOLEAN, ont déjà des prédicats ACL2 correspondants : `integerp`, `natp`, `complex-realp` et `booleanp`. Pour les autres types prédéfinis de VHDL : BIT, BIT_VECTOR, POSITIVE, il faut définir des nouveaux prédicats.

Dans ACL2 il existe la bibliothèque 'ihs' qui modélise les vecteurs de bits par des entiers, en considérant le bit de poids fort à droite. Cette modélisation a comme avantage la rapidité d'exécution. Par contre, les définitions des opérations sur les vecteurs de bit reposent sur les fonctions *mod* et *floor*. Généralement, en ACL2, il est assez difficile de raisonner sur ces fonctions. Même s'il existe aussi une bibliothèque spéciale pour *mod* et *floor*, souvent, dans la pratique, les règles ne suffisent pas. Un autre inconvénient de la bibliothèque est que les définitions considèrent le point de poids fort à droite. Aussi, une théorie sur les vecteurs de bits d'une taille fixe n'existe pas. Par exemple, 1 représente le bit 1, le vecteur de bit (1), le vecteur de bit (1 0), (1 0 0) etc.

Nous avons choisi de définir une bibliothèque qui traite les vecteurs de bits comme des listes. Cette bibliothèque est plus lente en vitesse d'exécution, mais il est plus facile de raisonner dessus, et elle reflète le comportement de vecteurs de bits en VHDL (le bit de

poids fort est à gauche). De même, puisque dans les circuits cryptographiques il existe de nombreuses manipulations de bits (rotation, concaténation) en plus des opérations arithmétiques, la description en termes de fonctions de la bibliothèque 'ihs', aurait compliqué la preuve.

Dans notre bibliothèque, le type BIT est défini par le prédicat `bitp` comme étant 0 ou 1. Le type BIT_VECTOR est reconnu par le prédicat `bit-vectorp` comme une liste d'éléments de type BIT. Pour l'instant, dans la traduction nous ne faisons pas de différence entre les types STD_LOGIC, UNSIGNED et BIT_VECTOR. Un vecteur de bit x de longueur n est reconnu par le prédicat `(wordp x n)`.

Dans VHDL, en plus des types prédéfinis, il existe aussi les types définis par l'utilisateur. Nous présentons par la suite comment les modéliser dans la logique ACL2. Pour chaque type nous décrivons la syntaxe VHDL, la spécification formelle de sa modélisation (définition et propriétés) et le code ACL2 correspondant. En effet, chaque type est modélisé par un prédicat qui reconnaît les éléments du type. Puisque les prédicats doivent prendre en compte tout type d'argument, de manière générale, nous considérons que la garde de la fonction ACL2 pour le prédicat est vraie.

Le type énumératif

Tout d'abord, analysons les types scalaires de VHDL. Le type énumératif a la forme suivante :

`type T is (`l_0`, ..., `l_{n-1}`) ;`

Pour modéliser le type `T`, nous générons :

- n définitions de symboles constants $*l_i* = i$ pour les éléments du type ;
- la définition de la constante $*T*$ égale à la liste des symboles l_i :$(l_0, \ldots, l_{n-1})$;
- un prédicat $T_p(x)$ qui réconnaît un élément x comme étant de type `T`, s'il appartient à l'ensemble $\{l_0, ..., l_n\}$:

$$T_p(x) = Member(x, *T*)$$

Puisque la définition du prédicat n'est pas récurrente, il faut définir des théorèmes sur le prédicat et ensuite interdire l'expansion de la fonction. Donc pour T_p est généré le théorème :

$$T_p\text{-rewrite} : T_p(x) \Rightarrow Member(x, *T*)$$

La figure 3.5 présente la modélisation ACL2 du type énumératif VHDL :
`type state is (s0,s1,s2,s3) ;`.

Le type entier borné

Un autre type scalaire est le type entier borné où `dir` est soit `to`, soit `downto`. :

`type T is range` a `dir` b `;`

```
(defconst *s0* 0)
(defconst *s1* 1)
(defconst *s2* 2)
(defconst *s3* 3)
(defconst *state*
          (list *s0* *s1* *s2* *s3*))

(defun statep (x)
   (declare (xargs :guard t))
   (member x *state* ))

(defthm statep-rewrite
  (implies (statep x)
     (member x *state* )))

(in-theory (disable statep))
```

FIG. 3.5 – Modèle ACL2 d'un type énumératif

Le type est modélisé par le prédicat $T_p(x)$ qui reconnaît qu'un élément x est de type `T` s'il est soit un entier dans l'intervalle $[a, b]$ si la direction est ascendante, soit un entier dans l'intervalle $[b, a]$ si la direction est descendante. Donc en fonction de la direction du type, nous générons l'un des deux prédicats :

$$T_p(x) = x \in \mathbb{Z} \ \wedge \ (a \leq x) \ \wedge \ (x \leq b), \text{ si } \texttt{dir} \text{ est } \texttt{to}.$$

$$T_p(x) = x \in \mathbb{Z} \ \wedge \ (b \leq x) \ \wedge \ (x \leq a), \text{ si } \texttt{dir} \text{ est } \texttt{downto}.$$

Comme dans le cas du type énumératif, il est indiqué de spécifier une règle de réécriture et désactiver ensuite la définition du prédicat :

$$T_p\text{-rewrite} : T_p(x) \Rightarrow x \in \mathbb{Z} \ \wedge \ (a \leq x) \ \wedge \ (x \leq b)$$

La Figure 3.6 montre la modélisation ACL2 pour le type VHDL :
`type WORD_INDEX is range 0 to 31 ;`.

Le type tableau borné

Maintenant, traitons le cas des types composés. Le type tableau borné a la forme suivante (où `dir` $\in$ {`to`, `downto`}) :

`type T_new is array (`a `dir` b`) of T ;`

Le type `T_new` est facilement modélisé comme une liste d'éléments de type `T`, de longueur

```
(defun  WORD_INDEXp (x)
    (declare (xargs :guard t))
    (and (integerp x)
         (<= 0 x)
         (<= x 31)))

(defthm WORD_INDEXp-rewrite
  (implies (WORD_INDEXp x)
     (and (integerp x)
         (<= 0 x)
         (<= x 31))))

(in-theory (disable WORD_INDEXp))
```

FIG. 3.6 – Modèle ACL2 d'un type entier borné

$|a-b+1|$. Ceci n'est pas suffisant, puisqu'il est important aussi de savoir dans quel intervalle se trouvent les indices du tableau et s'ils sont ascendants ou descendants, afin d'accéder correctement aux éléments du tableau. Par exemple l'élément indexé par $a+3$ est-il dans le tableau ? Oui, seulement si la direction du tableau est `to` et $a+3 \leq b$. Mais la position de l'élément dans la liste ACL2 qui modélise le tableau, n'est pas $a+3$, car en ACL2 la numérotation des éléments d'une liste l commence à 0, et continue jusqu'à $Len(l)-1$. Donc, dans ce cas, l'élément d'indice $a+3$ du tableau correspond à l'élément numéro 3 de la liste.

Ainsi, à la première vue, il faut stocker les deux bornes du tableau (ou une borne et la longueur) et la direction d'indices. En effet, il n'est pas nécessaire de connaître la direction. Pour tout tableau de longueur n, ascendant ou descendant, nous translatons les indices du tableau dans l'intervalle $[0, n-1]$ en calculant leur valeur relative à la borne gauche a du tableau. Ceci est réalisé par l'intermédiaire de la fonction *Calcul-pos*. Dans le corps de la fonction, $Left(tab)$ renvoie la borne gauche du tableau *tab*.

$$Calcul\text{-}pos(i, tab) = |Left(tab) - i|,$$

Finalement, il est suffisant de stocker seulement les bornes du tableau. Nous choisissons de stocker la borne gauche du tableau et la longueur, puisque cette information est souvent utilisée dans les preuves. Le prédicat qui reconnaît les éléments du type tableau borné `T_new` est :

$$T\_new_p(x) = List\_T_p(x) \ \wedge \ Len(x) = |a-b+1| \ \wedge \ Left(x) = a$$

où $List\_T_p$ est la définition d'une liste d'objets de type T. La liste vide est une liste valide. Si la liste n'est pas vide, alors le premier élément de la liste est de type `T` et la vérification continue sur le reste de la liste. Voici la spécification de $List\_T_p$:

$$List\_T_p(x) = \begin{cases} t, & \text{si } Empty(x) \\ T_p(First(x)) \wedge List\_T_p(Rest(x)), & autrement \end{cases}$$

Dans la définition du prédicat T_new_p apparaît aussi la fonction $Left$. Ceci est une fonction que nous devrons définir dans ACL2. Nous avons deux possibilités :

1. Définir une fonction *Left-tab* qui renvoie la valeur de la borne gauche pour chaque tableau *tab*. Dans ce cas il faut aussi définir une fonction *Calcul-pos-tab* pour chaque tableau. Ceci n'est pas très avantageux car par la suite il est difficile de décrire des propriétés générales pour le traitement des tableaux, et il faut à chaque fois traiter le cas particulier de chaque tableau.

2. Définir une fonction $Left$ unique pour tout tableau. Bien sûr cette possibilité est plus pratique, mais quel sera le corps d'une telle fonction ? ACL2 offre la possibilité, par l'encapsulation, de définir une fonction en sachant seulement sa signature, sans connaître son corps. Par encapsulation il est possible de définir une famille de fonctions avec des contraintes. Nous utilisons ce mécanisme pour définir la fonction $Left$ ayant un argument d'entrée et un de sortie. Pour la preuve, il est aussi utile de créer une règle qui donne le type de la fonction $Left$, qui est toujours un entier :

$$\forall x. Left(x) \in \mathbb{Z}$$

La condition pour que l'encapsulation soit acceptée est de fournir une fonction témoin qui respecte la signature et les contraintes définies. Comme fonction témoin, nous considérons la fonction $nfix(x)$ d'ACL2, définie pour tout objet et qui renvoie la valeur de x si x est un entier positif, sinon 0. La définition ACL2 de la fonction $Left$ est donnée de la manière suivante :

```
(encapsulate
   (((left *) => *))
   (local (defun left (x) (nfix x)))
   (defthm integerp_left
      (integerp (left x))
    :rule-classes ( :type-prescription :rewrite)))
```

La définition de T_new_p n'est pas récursive, donc nous générons des règles de réécriture pour la définition et nous la désaffectons par la suite. Nous générons trois règles, une pour chaque élément de la définition :

- la règle T_new_p-type spécifie que tout tableau est une liste des éléments de type `T`. Nous fixons le type de cette règle comme `forward-chaining`.

$$T\_new_p\text{-type} : \forall x.\ T\_new_p(x) \Rightarrow List\_T_p(x)$$

- la règle T_new_p-len spécifie la longueur du tableau.

$$T\_new_p\text{-len} : \forall x.\ T\_new_p(x) \Rightarrow Len(x) = |a - b + 1|$$

– la régle T_new_p-left spécifie la valeur de la borne gauche du tableau.

$$T\_new_p\text{-left} : \forall x.\ T\_new_p(x) \Rightarrow Left(x) = a$$

Les opérations les plus fréquentes sur un tableau sont la lecture et l'écriture de ses éléments. Dans ACL2 il existe déjà des bibliotheques avec des propriétés sur l'écriture et la lecture sur des listes (le tableau est une liste), mais il sera utile d'aider le démonstrateur avec un ensemble de théorèmes sur le type du résultat de ces deux opérations sur le tableau T_new :

– le théorème T_new_p-nth spécifie que le type d'un élément du tableau T_new est T_p. Nous fixons le type de cette règle comme `:rewrite` et `:type-prescription`.

$$T\_new_p\text{-nth:} \left.\begin{array}{l} \forall x.\ T\_new_p(x) \\ \forall i \in \mathbb{Z}.\ 0 \leq i < Len(x) \end{array}\right\} \Rightarrow T_p(Nth(i,x))$$

– le théorème T_new_p-update-nth spécifie qu'après la mise à jour d'un élément du tableau T_new avec une valeur de type T_p, le tableau reste bien défini. Cette règle s'encadre aussi dans les catégories `:rewrite` et `:type-prescription`.

$$T\_new_p\text{-update-nth} : \left.\begin{array}{l} \forall x.\ T\_new_p(x) \\ \forall i \in \mathbb{Z}.\ 0 \leq i < Len(x) \\ \forall val.T_p(val) \end{array}\right\} \Rightarrow T\_new_p(Update\text{-}Nth(i,val,x))$$

La Figure 3.7 présente la modélisation ACL2 du type VHDL :
`type MY_WORD is array (0 to 31) of BIT;`.

Le type tableau non borné

Le type tableau non borné a la forme :

```
type T_new is array (Z range <>) of T
```

Nous modélisons le type `T_new` par une liste d'éléments de type `T`, en utilisant le prédicat :

$$T\_new_p(x) = List\_T_p(x)$$

La définition de $List\_T_p$ est récursive et donc ne nécessite pas toutes les précautions prises pour les fonctions non récursives. Mais dans le même temps, il y a des propriétés qui sont utilisées de manière systématique dans les preuves sur les listes et que le démonstrateur aura aussi besoin de prouver pour $List\_T_p$. Afin d'aider l'outil dans son raisonnement, nous avons choisi de générer plusieurs théorèmes pour la définition $List\_T_p$ qui sont dans le même style que les théorèmes sur les tableaux bornés :

– le théorème $List\_T_p$-type spécifie que les tableaux non bornés sont de vraies listes.

```
(defun list-bitp (x)
  (declare (xargs :guard t))
  (if (true-listp x)
      (if (endp x) t
        (and (bitp (car x))
             (list-bitp (cdr x))))
    nil))

(defun MY_WORDp (x)
    (declare (xargs :guard t))
    (and (list-bitp x)
         (equal (left x) 0)
         (equal (len x) 32)))

(defthm MY_WORDp-type
  (implies (MY_WORDp x)
      (list-bitp x))
  :rule-classes :forward-chaining)

(defthm MY_WORDp-len
  (implies (MY_WORDp x)
      (equal (len x) 32)))

(defthm MY_WORDp-left
  (implies (MY_WORDp x)
      (equal (left x) 0)))

(defthm MY_WORDp-nth
  (implies (and (MY_WORDp x)
                (integerp x) (<= 0 i) (< i (len x)))
      (bitp (nth i x)))
  :rule-classes ( :type-prescription :rewrite))

(defthm MY_WORDp-update-nth
  (implies (and (MY_WORDp x)
                (integerp x) (<= 0 i) (< i (len x))
                (bitp val))
      (MY_WORDp (update-nth i val x)))
  :rule-classes ( :type-prescription :rewrite))

(in-theory (disable MY_WORDp))
```

FIG. 3.7 – Modèle ACL2 d'un type tableau borné

$$List\_T_p\text{-type} : \forall x.\ List\_T_p(x) \Rightarrow Listp(x)$$

– le théorème $List\_T_p$-car spécifie le type du premier élément du tableau.

$$List\_T_p\text{-car} : \forall x.\ List\_T_p(x) \wedge Cons(x) \Rightarrow T_p(First(x))$$

– le théorème $List\_T_p$-cdr spécifie que le reste du tableau après l'élimination du premier élément reste un tableau non borné.

$$List\_T_p\text{-cdr} : \forall x.\ List\_T_p(x) \wedge Cons(x) \Rightarrow List\_T_p(Rest(x))$$

– le théorème $List\_T_p$-nth spécifie que tout élément du tableau est de type `T`.

$$List\_T_p\text{-nth} : \left.\begin{array}{l} \forall x.\ List\_T_p(x) \\ \forall i \in \mathbb{Z}.\ 0 \leq i < Len(x) \end{array}\right\} \Rightarrow T_p(Nth(i,x))$$

– le théorème $List\_T_p$-update-nth spécifie qu'après la mise à jour d'un élément du tableau avec une valeur de type `T`, le tableau reste bien défini.

$$List\_T_p\text{-update-nth} : \left.\begin{array}{l} \forall x.\ List\_T_p(x) \\ \forall i \in \mathbb{Z}.\ 0 \leq i < Len(x) \\ \forall val.\ T_p(val) \end{array}\right\} \Rightarrow List\_T_p(Update\text{-}Nth(i,val,x))$$

La Figure 3.8 montre la modélisation du type VHDL :

```
type MEMORY is array (INTEGER range <>) of MY_WORD;
```

Le type enregistrement

Le dernier des types composés est le type enregistrement :

```
type T is
       record
          var0 : T0;
            ...
          varn : Tn;
       end record;
```

Pour modéliser le type enregistrement nous générons :
– $n+1$ symboles constants, un pour chaque champs du type var_i.
– un prédicat de type pour chaque T_i qui est un type entier contraint.
– un prédicat pour le type T qui reconnaît un élément x de type T si et seulement si x est une liste de $n+1$ éléments, et chaque élément à le bon type.

$$T_p(x) = Listp(x)\ \wedge\ Len(x) = n\ \wedge\ T_{0p}(x(0))\ \wedge \ldots \wedge\ T_{np}(x(n))$$

Puisque la définition de type n'est pas récursive, nous générons aussi les théorèmes liés au type, à la longueur, et au type de chaque champ du record :

```
(defun MEMORYp (x)
  (declare (xargs :guard t))
  (if (true-listp x)
      (if (endp x) t
        (and (MY_WORDp (car x))
             (MEMORYp (cdr x))))
    nil))

(defthm MEMORYp-type
  (implies (MEMORYp x)
      (true-listp x))
  :rule-classes :forward-chaining)

(defthm MEMORYp-car
  (implies (and (MEMORYp x) (cons x))
      (MY_WORDp (car x)))
  :rule-classes ( :type-prescription :rewrite))

(defthm MEMORYp-cdr
  (implies (and (MEMORYp x) (cons x))
      (MEMORYp (cdr)))
  :rule-classes ( :type-prescription :rewrite))

(defthm MEMORYp-nth
  (implies (and (MEMORYp x)
                (integerp x) (<= 0 i) (< i (len x)))
      (MEMORYp (nth i x)))
  :rule-classes ( :type-prescription :rewrite))

(defthm MEMORYp-update-nth
  (implies (and (MEMORYp x)
                (integerp x) (<= 0 i) (< i (len x))
                (MY_WORDp val))
      (MEMORYp (update-nth i val x)))
  :rule-classes ( :type-prescription :rewrite))
```

FIG. 3.8 – Modèle ACL2 d'un type tableau non-borné

$$T_p\text{-type} : \forall x.T_p(x) \Rightarrow Listp(x)$$

$$T_p\text{-len} : \forall x.T_p(x) \Rightarrow Len(x) = n + 1$$

$$T_p\text{-nth-i} : \forall x.T_p(x) \Rightarrow T_{ip}(x(i)), \text{ pour tout } 0 \leq i < Len(x).$$

Soit le type VHDL `DATE` :

```
type DATE is
     record
         DAY : INTEGER range 1 to 31 ;
         MONTH : INTEGER range 1 to 12 ;
         YEAR : INTEGER range 0 to 4000 ;
     end record ;
```

Le modèle ACL2 du type `DATE` est présenté dans la Figure 3.9.

3.2.2 Les déclarations de fonctions

Il existe deux types de déclarations de fonction dans un SER : les déclarations de fonctions définies dans le code VHDL, et les déclarations des fonctions créées pendant la transformation du code. Les fonctions de la première catégorie sont des fonctions définies par l'utilisateur et elles peuvent être récursives ou non. Les fonctions de la deuxième catégorie sont les fonctions récursives générées pour les boucles : for_x et $while\_x$.

Traitons tout d'abord les cas des fonctions définies par l'utilisateur. Le corps des fonctions VHDL a été transformé (conformément au chapitre 3) en expressions. Donc la déclaration d'une fonction a la forme :

```
function nom (v_1 : T_1, ...,  v_n : T_n) return T is
       begin
         return expression ;
       end function ;
```

Modéliser une déclaration de fonction en ACL2 revient à créer deux événements :

- l'événement `defun` suivant, où la fonction `prefix` transforme l'expression `expression` en une forme prefixée :

```
(defun nom (v_1 ... v_n)
       (declare (xargs :guard (and (T_1p v_1)
                                   ...
                                   (T_np v_n))))
       (prefix expression))
```

- le théorème de type du résultat de la fonction : `nom-type`. Le théorème est de type `:rewrite` et `:type-prescription`.

```
(defconst *DAY* 0)
(defconst *MONTH* 1)
(defconst *YEAR* 2)

(defun DAYp (x)
  (declare (xargs :guard t))
  (and (integerp x) (<= 1 x) (<= x 31)))

(defun MONTHP (x)
  (declare (xargs :guard t))
  (and (integerp x) (<= 1 x) (<= x 12)))

(defun YEARp (x)
  (declare (xargs :guard t))
  (and (integerp x) (<= 0 x) (<= x 4000)))

(defun DATEp (x)
  (declare (xargs :guard t))
  (and (true-listp x) (equal (len x) 3)
       (DAYp (nth *DAY* x))
       (MONTHp (nth *MONTH* x))
       (YEARp (nth *YEAR* x))))

(defthm DATEp-type (x)
  (implies (DATEP x) (true-listp x))
  :rule-classes :forward-chaining)

(defthm DATEp-len
  (implies (DATEp x) (equal (len x) 3)))

(defthm DATEp-nth-0
  (implies (DATEp x) (DAYp (nth *DAY* x)))
  :rule-classes ( :type-prescription :rewrite))

(defthm DATEp-nth-0
  (implies (DATEp x) (MONTHp (nth *MONTH* x)))
  :rule-classes ( :type-prescription :rewrite))

(defthm DATEp-nth-0
  (implies (DATEp x) (YEARp (nth *YEAR* x)))
  :rule-classes ( :type-prescription :rewrite))
```

FIG. 3.9 – Modèle ACL2 d'un type enregistrement

$$\texttt{nom-type}: \left.\begin{array}{l} \forall v_1.T_{1p}(v_1) \\ \ldots \\ \forall v_n.T_{np}(v_n) \end{array}\right\} \Rightarrow T_p(nom(v_1,\ldots,v_n))$$

Pour les fonctions non récursives, l'acceptation de l'événement par ACL2 est presque triviale. Le seul endroit où des difficultés peuvent apparaître est la vérification de gardes, si les domaines des paramètres sont plus larges que l'utilisation effective dans la fonction. Dans ce cas, l'utilisateur doit rendre les gardes plus précises ou les éliminer complètement. Pour la même raison, il est possible que les hypothèses du théorème ne soient pas suffisantes, et dans ce cas l'acceptation du théorème échoue. L'utilisateur a bien sûr la possibilité de corriger la définition des hypothèses, pour que le théorème soit vrai.

Dans le cas des fonctions récursives, le démonstrateur doit prouver que la fonction a une mesure décroissante. Si la fonction récursive n'a pas de condition d'arrêt, alors il y a une erreur de conception et la définition ne sera pas acceptée par ACL2. Si la fonction est bien construite, alors le démonstrateur va essayer de prouver sa terminaison. Le problème est que beaucoup de fonctions récursives correctes du point de vue du programmeur ne sont pas admissibles par ACL2.

Prenons par exemple la fonction $g : \mathbb{N} \to \mathbb{N}$:

$$g(x) = \begin{cases} 0 \text{ si } x = 0 \\ g(x-1) \; autrement \end{cases}$$

Le code VHDL de la fonction est :

```
function g (x : NATURAL) return NATURAL is
 begin
   if x=0 then return 0 ;
     else return g(x-1) ;
   end if ;
 end function g ;
```

Après transformation, la déclaration de g devient :

```
function g (x : NATURAL) return NATURAL is
 begin
   return IF(x=0, 0, g(x-1)) ;
 end function g ;
```

Donc, l'événement `defun` associé est :

```
(defun g (x)
   (declare (xargs :guard (natp x)))
   (if (equal x 0) 0
      (g (- x 1))))
```

La définition de g est partielle, car g s'applique seulement aux entiers positifs, mais en ACL2 toutes les fonctions sont totales. Les gardes n'ont pas une valeur logique, étant prises en compte seulement pour l'exécution de la fonction. Ainsi, la définition de g n'est pas admissible par ACL2, puisque le principe de définition n'est pas respecté : il n'existe pas de mesure ordinale de x qui décroît dans l'appel récursif (par exemple le calcul de `g(-1)` ne s'arrête pas).

Manolios et Moore [84] ont défini un mécanisme basé sur l'encapsulation pour accepter les fonctions partielles en ACL2. Il existe trois méthodes pour introduire une fonction partielle dans la logique ACL2 : soit la fonction partielle est récursive terminale, soit la fonction partielle est fournie avec une fonction témoin totale, soit une mesure décroissante est définie pour un domaine contraint de la fonction partielle. En effet, dans tous les cas, une fonction témoin totale qui respecte le principe de définition est générée. La fonction partielle est considérée ensuite comme une instance contrainte de la fonction témoin. Techniquement, ceci est réalisé par l'intermédiaire de la macro `defpun`.

```
(defpun g (x)
   (if (equal x 0) 0
      (g (- x 1))))
```

L'événement précédent est du sucre syntaxique pour la forme :

```
(encapsulate
   (((g * ) => * ))
   (local (defun g(x) ...))
   (defthm g-def
       (equal (g x)
              (if (equal x 0) 0
                  (g (- x 1))))
       :rule-classes :definition))
```

La principale contribution de [84] est de montrer que pour les fonctions récursives terminales il est possible de générer automatiquement la définition de la fonction locale $g(x)$.

Puisque la fonction partielle est encapsulée, elle a un axiome de définition associé dans la logique, mais elle n'a pas un corps défini et donc, n'est pas exécutable. Ceci est un inconvénient de la macro `defpun`, puisque dans certains cas il est utile d'évaluer l'appel de la fonction sur des valeurs constantes. Pour répondre à ce problème, Ray [106] définit une autre macro `defpun-exec` qui permet d'introduire pour une fonction partielle définissable avec `defpun`, une contrepartie exécutable en Lisp. La macro `defpun-exec` tient compte aussi de gardes pour la fonction partielle. La forme générale de la macro est :

`(defpun-exec` f `(`v_1 ... v_n`)` β `:guard` $formula$`)`

De point de vue logique, i.e. de la preuve, la macro `defpun-exec` a le même effet que `defpun`.

Pour la définition de toutes les fonctions récurrentes définies par l'utilisateur, nous allons utiliser la macro `defpun-exec`. L'événement suivant est maintenant accepté par le démonstrateur.

```
(defpun-exec g (x)
   (if (equal x 0) 0
      (g (- x 1)))
    :guard (natp x))
```

Dans le cas des fonctions introduites par la transformation de code, nous utilisons aussi le macro `defpun-exec`. Puisque les fonctions for_x et $while\_x$ sont récursives terminales, nous avons la garantie qu'elles sont acceptées par le démonstrateur, sans devoir fournir une fonction témoin ou une mesure particulière.

3.2.3 Les déclarations d'objets

Les déclarations des objets d'un SER, T_I, T_L et T_S, définissent l'appartenance d'un objet à un type. La déclaration d'un objet à la forme `x : T` ou `x : T(a dir b)`. Donc, dans la logique ACL2 ceci est équivalent à dire :

- pour le premier cas, la valeur de vérité de la formule T_p(x) est vrai.
- dans le deuxieme cas, la valeur de vérité de la formule $T_p(x) \wedge Len(x) = |a - b + 1| \wedge Left(x) = a$ est vrai.

T_p est le prédicat de type pour T, défini conformément à la section précédente.

3.2.4 Les équations récurrentes

Pour toute équation récurrente $x(t+1) := f_x(y_1(t), \ldots, y_n(t))$ nous définissons une fonction ACL2 non récursive :

```
(defun nextval_x (y1 ... yn)
    (declare (xargs :guard (and (T_y1p y1)
                                ...
                                (T_ynp yn))))
    (prefix expression))
```

La fonction $nextval\_x$ modélise une itération de l'exécution de l'équation. *expression* est soit une valeur constante, soit un symbole, soit un appel de fonction. T_{y_ip} est le prédicat qui reconnaît le type de l'argument y_i.

Les valeurs constantes de type scalaire de VHDL ont leur correspondant direct en ACL2 (pour les types entier, caractère, réel). Les valeurs constantes de type vecteur de bit sont transformées en listes en ACL2. Par exemple "010" devient la liste `(0 1 0)`. Les différentes bases de calcul sont aussi prises en compte. Nous avons défini des fonctions qui

transforment un vecteur de bits de base 16, ou 8 vers un vecteur de base 2. Donc O"010" devient `(octal (0 1 0))`, et X"010" devient `(hexa (0 1 0))`.

Les symboles identificateurs pour variables, constantes, signaux, restent les mêmes en ACL2.

Généralement, l'appel de fonction $f(a_1, \ldots, a_n)$ devient (f a_1 ... a_n) et les expressions arithmétiques et booléennes sont écrites selon la forme préfixée.

A présent, les fonctions et les opérateurs (arithmétiques, booléens, et de relation) sont ceux définis en VHDL ou ceux définis pendant le processus de transformation. Il faut donc les associer avec des fonctions ACL2.

Ceci n'est pas trivial, car VHDL supporte la surcharge des fonctions, tandis que ACL2 non. Afin de garder le processus de traduction le plus automatique possible, nous avons choisi de définir une seule fonction ACL2 pour toutes les variantes VHDL de la même fonction. Ainsi, un symbole VHDL est remplacé par un symbole ACL2 systématiquement, sans faire une analyse de type, qui d'ailleurs sera très difficile (voire impossible) dans un environnement complètement symbolique. Cette approche est efficace pour les fonctions dont les surcharges ont des types de paramètres différents mais le même nombre de paramètres. Ceci est une restriction du sous-ensemble VHDL que nous prenons en compte. Si la surcharge d'une fonction a une signature différente en nombre de paramètres de la fonction initiale, alors nous ne la traitons pas. Dans ce cas, nous demandons l'utilisation d'un autre nom pour la nouvelle fonction, ainsi la surcharge est éliminée. Voici un exemple de modélisation de la fonction '+' surchargée pour deux paramètres de type bit vecteur (premier cas), un vecteur de bits (non signé) et un naturel (deuxième cas), un naturel et un vecteur de bits (non signé) (troisième cas). Si le type de paramètres de la fonction ne corresponde à aucun des types considérés auparavant, alors la fonction '+' d'ACL2 est invoquée (quatrième cas).

```
(defun binary-plus (x y)
  (declare (xargs :guard t))
  (cond ((and (bvp x) (bvp y))
         (let ((i (max (len x) (len y))))
           (bv-to-n (int-bv-be
                      (+ (bv-int-be x)
                         (bv-int-be y)))
                    i)))
        ((and (bvp x) (natp y))
         (bv-to-n (int-bv-be
                    (+ (bv-int-be x) y))
                     (len x)))
        ((and (natp x) (bvp y))
         (bv-to-n (int-bv-be
                    (+ x (bv-int-be y)))
                     (len y)))
```

VHDL	**ACL2**
and	b-and
or	b-or
xor	b-xor
not	b-not
+	plus
-	minus
*	b-*
/	/
rem	floor
mod	mod
&	append
**	expt
=	b-equal
\=	(not (b-equal ...))
<	b-<
<=	b-<=
>	b->
>=	b->=
CONV_INTEGER	bv-nat-be
CONV_UNSIGNED	nat-bv-be
ext	bv-to-n
shl	shl
shr	shr

FIG. 3.10 – Des opérateurs VHDL et leurs correspondants ACL2

```
(t (+ x y))))
```

Pour compléter la modélisation de VHDL, nous avons pris en compte les bibliothèques standard VHDL : std_logic_1164 et numeric_std. En effet, au lieu de considérer le type STD_LOGIC avec ses 9 valeurs, nous considérons seulement le type BIT_VECTOR avec les deux valeurs 0 et 1, reconnu par le prédicat ACL2 `bit-vectorp`. Le type UNSIGNED est aussi reconnu par le même prédicat.

Nous avons donc modélisé en ACL2 des opérations logiques (et, or, xor, not) entre deux ou plusieurs vecteurs de bit, l'opération de conversion d'un vecteur de bit vers un entier positif, l'opération d'extension d'un entier vers un vecteur de bit d'une longueur donnée, des opérations arithmétiques (+, -) entre deux vecteurs de bits ou un entier et un vecteur de bit, des opérations de rotation à gauche ou à droite. La Figure 3.10 montre la correspondance entre quelques opérateurs VHDL et nos fonctions ACL2. Nous avons implementé ces opérateurs pour les types BIT_VECTOR, UNSIGNED et INTEGER et pour les combinaisons possibles, telles que définies dans les bibliotheques standard std_logic_1164, numeric_std et les bibliotheques STD_LOGIC_ARITH et STD_LOGIC_UNSIGNED.

Notre bibliothèque est facilement extensible. Si par exemple, l'on veut ajouter le type SIGNED et l'addition entre un élément SIGNED et un entier, il faut seulement ajouter un autre cas dans la définition de notre fonction *plus*. Tous les théorèmes prouvés auparavant resteront vrais, puisque leurs hypothèses concernaient les autres cas de la définition. Il faudra définir de nouvelles propriétés correspondant au comportement de la fonction dans le nouveau cas ajouté.

Maintenant traitons le cas des fonctions que nous avons défini pendant le processus de transformation de code. Les fonctions concernées sont la fonction IF, *write*, *read* et les fonctions for_x et $while\_x$. La fonction IF a comme correspondant direct le constructeur `(if ...)` d'ACL2. Les fonctions for_x et $while\_x$ sont définies dans la transformation, et leur définition est traduite en ACL2 conformément à la section précédente.

La fonction *write* prend en entrée une liste de couples (*pos*, *val*) et un objet et retourne l'objet après que chaque valeur d'élément de position *pos* a été remplacée par *val*. L'écriture de l'objet se réalise de l'intérieur vers l'extérieur, de sorte que seule la dernière écriture compte. Par exemple, le résultat de l'appel *write* ((((2), 10), ((1), 5), ((2), 3)), (0 1 2 3)) est (0 5 10 3).

Nous rappelons que la fonction a été créée pour modéliser les écritures successives dans un tableau. Dans l'exemple précédent, le premier argument de la fonction, la liste (((2), 10),((1), 5), ((2), 3)), est une liste de couples où (1) et (2) sont des positions à écrire dans l'objet et 10, 5 et 3 sont des valeurs de remplacement. Le deuxième argument de la fonction, (0 1 2 3), est l'objet à modifier. Si l'objet à écrire est un objet multidimensionnel, alors le type des positions n'est plus un entier, mais une liste d'entiers. Par exemple pour écrire l'élément de la 2ème ligne, et 3ème colonne d'une matrice, la position de l'objet est la liste (2,3) (nous supposons que la numérotation débute à 0). Un cas plus complexe est l'écriture d'une partie d'un tableau : (a dir b). Nous rappelons que dans ce cas la position est la liste (dir a b).

Voici une implémentation possible de *write*. Si la liste des écritures est vide, alors la fonction retourne l'objet. Sinon, si la première position d'écriture est vide, la valeur à écrire est retournée. Sinon, la première écriture de la liste (qui est en effet la dernière écriture physique) est effectuée après que toutes les autres écritures de la liste ont été effectuées (la fonction $modify$).

$$write(L,x) = \begin{cases} x \text{ si } Empty(L) \\ expr, \text{ si } First(L) = (pos, expr) \wedge Empty(pos) \\ modify(pos, val, write(Rest(L), x)), \text{ autrement} \end{cases}$$

La fonction $modify$ prend comme entrée un objet x, la position *pos* (qui est une liste) à écrire dans l'objet et une valeur *val*, et retourne l'objet après l'écriture. Si la position est vide, alors il n'y a pas d'écriture à réaliser, et donc l'objet est retourné. Sinon, nous avons deux cas :

- Si la position est une liste de type (dir *a b*), alors les éléments de *a* à *b* de l'objet sont écrits avec la valeur *val*.
- Sinon, l'élément de la première position dans la liste, est mis à jour avec la valeur de

l'élément après l'ecriture.

$$modify(pos, val, x) = \begin{cases} x, \text{ si } Empty(pos) \\ \\ Append(Firstn(a, x), val, Restn(b, x)), \\ \quad \text{si } pos = (dir, a, b) \\ \\ Updateth(First(pos), \\ \qquad\qquad modify(Rest(pos), val, Nth(First(pos), x)), \\ \qquad\qquad x), \\ \quad autrement \end{cases}$$

La fonction *read* prend comme entrée une position *pos* et un objet x et retourne la valeur de l'élément de la position *pos*. Si la position est la liste vide, la fonction retourne la valeur de l'objet x. Sinon, elle retourne la valeur qui se trouve sur la position *pos* de la valeur de l'objet x. Une position est toujours une liste d'entiers ou une liste de type (dir a b). Par exemple, l'appel *read* ((1, (to, 0, 1)), ((0, 1), (2, 3), (4, 5))) est évalué à (2, 3).

Voici une implémentation possible de *read*.

$$read(pos, x) = \begin{cases} x, \text{ si } Empty(pos) \\ read(Rest(pos), Segment(a, b, x)) \text{ si } First(pos) = (dir, a, b) \\ read(Rest(pos), Nth(First(pos), x)), \text{ autrement} \end{cases}$$

Remarquons que les définitions de *read* et *write* considèrent que pour un tableau de longueur k, l'indice du tableau prend des valeurs de 0 à $k-1$. Ceci n'est pas nécessairement vrai dans VHDL. Nous pouvons même dire que dans la majorité des descriptions matérielles les tableaux sont décroissants, débutant avec la valeur $k-1$, et descendant à 0. Dans la transformation de code VHDL vers un SER, nous nous sommes contentés de stocker les indices tels qu'ils étaient utilisés en VHDL. Pour avoir une modélisation correcte en ACL2 il faut donc associer à chaque indice de tableau sa valeur relative à une échelle croissante positive, débutant à 0. En effet, pour tout tableau qui débute à n, la valeur relative d'un indice i du tableau par rapport à notre échelle est la valeur absolue de la différence entre n et i : *n - i*. Nous rappelons que dans notre modélisation des types tableau nous avons défini une fonction $Left$ qui stocke la valeur de début pour les indices du tableau et la fonction $Calcul-pos(i, x)$ qui pour tout objet et indice VHDL, calcule la valeur de l'indice ACL2 : *Left*(x) - *i*. Puisque les fonctions *read* et *write* prennent comme entrées des positions qui sont les listes des indices, nous avons défini des fonctions qui font correspondre aux positions VHDL leurs positions ACL2 valides.

Le code ACL2 pour les fonctions de calcul des nouveaux indices se trouve dans la Figure 3.11 et le code ACL2 pour les fonctions *write*, $modify$, *read* se trouve dans la Figure 3.12. Puisque les mots *read* et *write* sont des mots réservés en ACL2, nous utilisons les noms `read-elem` et `write-elem` pour les fonctions ACL2.

Donc l'appel de la fonction *write*(*liste-pos-val*, x) dans le SER, devient

`(write-elem (calc-list-pos-val liste-pos-val x) x)` en ACL2.

De manière similaire, l'appel de la fonction *read*(*liste-pos*, x) dans le SER, devient

`(read-elem (calc-list-pos liste-pos x) x)` en ACL2.

3.2.5 Le système

Soit $SER_P = \langle \mathcal{I}n, \mathcal{L}ocals, \mathcal{O}ut, Init, \{E_x : x(t+1) = f_x(y(t), \ldots)\}_{x \in \mathcal{L}ocals \cup \mathcal{O}ut}, Decl \rangle$, le système correspondant à la description P. Nous avons au moins deux alternatives pour modéliser le système dans son intégralité :

1. Par des équations mutuellement récurrentes.
2. Par une fonction *step* qui calcule l'exécution d'un pas pour toutes les équations.

La première variante semble la plus naturelle. Nous utilisons la commande ACL2 `mutual-recursion` et chaque objet du SER devient une fonction dépendant du temps.

Donc, pour tout objet x de $\mathcal{L}ocals \cup \mathcal{O}ut$, pour lequel il existe une équation $x(t+1) := f_x(y_1(t), \ldots, y_n(t))$, est générée une fonction (t est remplacé par n, puisque t est un mot réservé en ACL2) :

```
(defun x (n)
   (declare (xargs :guard (integerp n)))
   (if (zp n) (init x)
       (nextval_x (y1 (- n 1)) ... (y1 (- n 1))))
```

`nextval_x` est le correspondant ACL2 de la fonction f_x de l'équation récurrente de x.

`(init x)` est l'expression associée à l'objet x dans la phase d'initialisation, c'est à dire par $Init$. S'il n'existe pas d'expression associée explicitement à x dans $Init$, nous définissons une fonction contrainte de retourner un objet du même type que x :

```
(encapsulate
    (((init * ) => * )
     ((type-initp * ) => * ))
    (local (defun init (x) (declare (ignore x)) 0))
    (local (defun type-initp (x) (integerp x)))
    (defthm init-typep
            (type-initp (init x))))
```

Pour les objets d'entrée il n'existe pas de définition, mais pour que le système soit bien construit, il faut créer une fonction par objet. Nous définissons par l'encapsulation, des fonctions dépendantes de n et contraintes d'avoir le même type que l'objet d'entrée. Soit $\mathcal{I}n = \{in_1, \ldots, in_k\}$. L'événement suivant est généré :

```
(encapsulate
```

```
(defun calcul-pos (indice x)
  (declare (xargs :guard (integerp indice)))
  (abs (- (left x) indice)))

(defun calc-list-pos (liste-pos x)
  (declare (xargs :guard (and (true-listp x)
                              (list-posp liste-pos))))
  (if (endp liste-pos) nil
    (if (atom (car liste-pos))
        (cons (calcul-pos (car liste-pos) x)
              (calc-list-pos (cdr liste-pos) x))
      (cons (list (nth 0 (car liste-pos))
                  (calcul-pos (nth 1 (car liste-pos)) x)
                  (calcul-pos (nth 2 (car liste-pos)) x))
            (calc-list-pos (cdr liste-pos) x)))))

(defun calc-list-pos-val (liste-pos-val x)
  (declare (xargs :guard (and (true-listp x)
                              (list-pos-valp liste-pos-val))))
  (if (endp liste-pos-val) nil
    (if (and (consp liste-pos-val)
             (endp (cdr liste-pos-val)))
        (cons (calcul-pos (get-first-position liste-pos-val) x)
              (calc-list-pos-val (cdr liste-pos-val) x))
      (cons (list (nth 0 (get-first-position liste-pos-val))
                  (calcul-pos (nth 1 (get-first-position
                                      liste-pos-val)
                                    x))
                  (calcul-pos (nth 2 (get-first-position
                                      liste-pos-val)
                                    x)))
            (calc-list-pos-val (cdr liste-pos-val) x)))))
```

FIG. 3.11 – Modèle ACL2 des fonctions de calcul des indices VHDL

```
(defun modify (pos val x)
  (declare (xargs :guard (and (true-listp x)
                              (posp pos))))
 (let ((p (car pos)))
    (if (endp pos) x
      (if (or (equal p 'to)
              (equal p 'downto))
        (append (firstn (nth 1 pos)  x) val (nthcdr (nth 2 pos) x))
      (update-nth p (modify (cdr pos) val (nth p x)))))))

(defun write-elem (liste-pos-val x)
  (declare (xargs :guard (and (true-listp x)
                              (list-pos-valp liste-pos-val))))
  (let ((p (get-first-position liste-pos-val))
        (v (get-first-value liste-pos-val)))
    (if (endp liste-pos-val) x
      (if (endp p) v
      (modify p  v (write-elem (cdr liste-pos-val) x))))))

(defun read-elem (liste-pos x)
  (declare (xargs :guard (and (true-listp x)
                              (list-posp liste-pos))))
  (if (endp liste-pos) x
    (if (atom (car liste-pos))
        (read-elem (cdr liste-pos) (nth  (car liste-pos) x))
      (read-elem (cdr liste-pos)
            (segment (get-a liste-pos)
                     (get-b liste-pos) x)))))
```

FIG. 3.12 – Modèle ACL2 des fonctions *read* et *write*

```
(((in1 * ) => * )
 ((type_in1 * ) => * )
 ...
 ((ink * ) => * )
 ((type_ink * ) => * ))
(local (defun in1 (n) (declare (ignore n)) 0))
(local (defun in1p (x) (integerp x)))
(defthm type-in1
        (in1p (in1 x)))
... )
```

A présent le système est complet. L'utilisateur peut instancier les types des objets et peut définir des hypothèses sur les fonctions d'entrée du SER. Cette modélisation a quelques inconvenients : le plus important est qu'il n'existe pas beaucoup d'heuristiques pour gérer la récursivité mutuelle. Donc les preuves sur le système ne sont pas faciles. Un autre inconvénient est la vitesse d'exécution du système.

L'autre alternative de modélisation du système est par une fonction récurrente. Un appel de la fonction correspond au calcul de la fonction *step* qui représente l'exécution simultanée d'un pas de toutes les équations récurrentes.

Dans cette modélisation, nous définissons un symbole ACL2 pour tout objet du SER. Soit $\mathcal{I}n = \{in_1, \ldots, in_k\}$, $\mathcal{L}ocals = \{l_1, \ldots, l_m\}$, et $\mathcal{O}ut = \{o_1, \ldots, o_p\}$. Nous définissons la fonction *step* qui prend comme argument la liste des symboles d'entrée, *in*, et la liste concaténée des symboles locaux et de sortie, *st*. Tout d'abord, nous définissons des constantes qui donnent la position de chaque symbole dans la liste *in* ou *st*. IL s'agit donc de $k+m+p$ définitions de constantes :

```
(defconst *in1* 0)
...
(defconst *ink* k-1)
(defconst *l1* 0)
...
(defconst *lm* m-1)
(defconst *o1* m)
...
(defconst *op* m+p-1)
```

La fonction *step* retourne la liste des valeurs calculées par les fonctions des équations récurrentes (f_x) dans leur variante ACL2 (`nextval_x`).

```
(defun step (in st)
  (let ((in1 (nth *in1* in))
        ...
        (ink (nth *ink* in))
```

```
      (l1  (nth *l1* st))
      ...
      (op (nth *op* st)))
  (list (nextval_l1 ...)
        ...
        (nextval_op ...))))
```

La fonction *step* n'est pas une fonction récursive, donc nous générons un théorème de réécriture pour la fonction, et ensuite nous désactivons sa définition :

$$\text{step-rewrite} : step(in, st) \Rightarrow List(nextval\_l1(...), \ldots, nextval\_op(...))$$

Afin de faciliter le raisonnement sur le système nous générons aussi deux prédicats :

- le prédicat de type des entrées, reconnaît une entrée bien construite si chaque élément a bien son type tel que dans la déclaration VHDL.

$$Hyp\_input(in) = T_{in_1 p}(in_1) \wedge \ldots \wedge T_{in_k p}(in_k)$$

- le prédicat de type de l'état, reconnaît un état bien construit si chaque élément a le type de sa déclaration VHDL.

$$Hyp\_st(st) = T_{l_1 p}(l_1) \wedge \ldots \wedge T_{o_p p}(ip_p)$$

Maintenant il est possible de prouver le théorème de type pour la fonction *step* :

$$\text{step-type} : \forall in.Hyp\_in(in) \wedge \forall st.Hyp\_st(st) \Rightarrow Hyp\_st(step(in, st))$$

Le système complet est la fonction récurrente *system* définie sur une liste d'entrées de longueur arbitraire et un état. S'il n'y a plus de stimulus d'entrée, la fonction renvoie l'état du système. Sinon, la fonction *step* est appelée pour les premières entrées de la liste, et le calcul continue avec le nouvel état du système et le reste d'entrées.

```
(defun system (list-in st)
   (if (atom list-in) st
     (system (cdr list-in) (step (car list-in) st))))
```

Pour la fonction *system* il faut aussi générer un théorème de type. Pour cela il est nécessaire de définir le type de `list-in`, qui est une liste des éléments du même type : *Hyp_input*. Le théorème `system-type` donne le type de la fonction *system*.

$$\text{system-type} : \forall list\text{-}in.List\text{-}Hyp\_in(list\text{-}in) \wedge \forall st.Hyp\_st(st) \Rightarrow Hyp\_st(system(list\text{-}in, st))$$

La fonction *system* ainsi définie a une propriété qui permet la composition du comportement du circuit.

$$\forall L1, L2. \forall st. system(L1, (system L2, st)) = system(Append(L1, L2)st)$$

Conclusion

Dans ce chapitre nous avons présenté le démonstrateur ACL2 et la manière dont un système d'équations récurrentes est traduit dans sa logique. Nous avons montré comment générer des définitions, mais aussi des théorèmes, afin de diminuer l'effort de l'utilisateur pour la modélisation du système matériel, mais aussi pour la preuve des propriétés. Malheureusement ceci n'est pas suffisant, une grande partie de l'effort de preuve reste à faire.

L'idée de décrire un système comme une fonction récurrente qui modifie l'état du système n'est pas nouvelle [93], elle a été déjà utilisée dans la communauté de la démonstration de théorèmes. Notre intention a été de traduire d'une manière efficace une description matérielle vers un tel modèle.

Une première modélisation d'un langage de description matérielle en ACL2 a été définie par Russinoff [113], pour un sous-ensemble très réduit d'un HDL propre. Dans [47], Georgelin modélise un sous-ensemble réduit de VHDL. Le système est toujours modélisé par une fonction récurrente. Un pas de la fonction est réalisé par une fonction ACL2 qui est la traduction directe du code VHDL dans ACL2. La simulation symbolique de la description est réalisée dans ACL2. Cette manière de représenter le système s'est avéré inefficace, à cause de la dimension et de la forme de la fonction qui représente le système.

Nous apportons deux améliorations à ces travaux. D'une part nous traitons un sous-ensemble plus large de VHDL, qui inclut les types composés, les sous-programmes, les boucles et les processus avec plusieurs instructions de synchronisation. En plus, nous n'imposons pas des règles d'écriture pour la reconnaissance syntaxique de l'horloge. D'autre part le modèle que nous proposons est plus modulaire, et la manière de le décrire en ACL2 est plus adaptée à la preuve par récurrence.

Dans le chapitre suivant, nous présentons la problématique de vérification des circuits numériques cryptographiques. Nous avons modélisé nos études de cas en ACL2, d'après la méthode présentée précédemment, et nous avons prouvé que le circuit implémente correctement l'algorithme de spécification.

Chapitre 4

Vérification formelle des composants cryptographiques

4.1 Introduction à la cryptographie

La cryptographie a une longue et fascinante histoire [69]. La première utilisation documentée de la cryptographie est datée de 1900 B.C., quand un scribe égyptien utilise des hiéroglyphes dans une inscription d'une manière différente de l'écriture classique. Depuis, la cryptographie a été utilisée plutôt comme art, généralement pour garder des secrets nationaux et de stratégie. A partir des années '60, l'art devient science, le développement des ordinateurs et des réseaux de communication créant une nouvelle demande de protéger l'information en format numérique, et d'avoir des services qui garantissent la sécurité des transmissions de données.

La cryptographie est donc aujourd'hui une science qui étudie les techniques mathématiques liées aux aspects de sécurité de l'information comme la confidentialité, l'intégrité des données, l'authentification des parties et l'authentification de l'origine de données (ou non répudiation). La cryptographie propose des mécanismes afin de garantir ces aspects le plus longtemps possible.

Il existe trois types de schémas cryptographiques utilisés pour assurer la sécurité :

- la cryptographie à clé privée (ou symétrique)
- la cryptographie à clé publique (ou asymétrique)
- les fonctions de hachage

La cryptographie à clé privée utilise une seule clé pour le cryptage et le décryptage (une autre possibilité est que la clé pour le décryptage soit facilement obtenue à partir de la clé de cryptage). Les algorithmes les plus connus de ce type sont DES, avec sa variante TDES et AES.

La cryptographie à clé publique utilise une clé publique pour le cryptage et une clé privée pour le décryptage. Le calcul de la clé privée à partir de la clé publique est très difficile, pratiquement impossible. Un algorithme connu pour le cryptage à clé publique est RSA. Généralement, ce type de cryptage est utilisé pour la signature numérique.

Les fonctions de hachage utilisent une transformation mathématique pour crypter le

message de manière irréversible. Elles sont utilisées en combinaison avec les deux autres types de schémas cryptographiques : à clé publique et à clé privée, pour garantir l'authentification du message et l'intégrité de son contenu. Les fonctions de hachage les plus utilisées sont SHA et MD5.

Les composants cryptographiques se trouvent aujourd'hui dans presque toutes les puces électroniques, donc pour garantir la sécurité de l'information, il est important de garantir que :

1) l'algorithme de cryptage respecte les critères de sécurité,

2) l'implémentation matérielle des composants est correcte par rapport à l'algorithme.

Dans notre travail, nous nous intéressons au deuxième point en utilisant la démonstration de théorèmes.

4.2 Présentation générale de l'approche

Afin de prouver qu'une implémentation est correcte par rapport à sa spécification, il faut modéliser les deux aspects (implémentation et spécification) dans la logique de l'outil de démonstration.

La spécification est généralement un ensemble de textes en anglais ou français, avec des diagrammes et des morceaux d'algorithme. Cette spécification informelle est transformée dans un modèle formel qui devient le modèle de référence pour l'étape de vérification. Soit $Spec(params)$ un tel modèle, où $params$ représente les paramètres de la spécification. Idéalement, des propriétés de correction du modèle sont prouvées afin de garantir sa bonne construction.

L'implémentation, de notre point de vue, est un circuit décrit en VHDL. Nous traduisons le circuit dans un modèle formel ACL2, conformément à la méthode exposée dans cette thèse. Soit $System(Input, st)$ le modèle obtenu, où $Input$ représente une série de valeurs pour les entrées du circuit et st représente l'ensemble des valeurs des objets mémorisants et de sortie du circuit.

Maintenant, nous avons deux modèles dans la logique d'ACL2 : $Spec$, obtenu à la main, à partir de la spécification, et $System$, obtenu automatiquement à partir de la description VHDL. Nous voulons prouver que le deuxième modèle implémente correctement le premier. Tout d'abord, il faut décrire cette propriété dans la logique du démonstrateur. Généralement, il n'existe pas de relation directe entre les deux modèles, puisque dans l'implémentation il y a plus de détails de temps, de structure, de comportement, et les données sont raffinées.

De manière informelle, pour prouver qu'un circuit implémente une spécification, on veut prouver qu'à partir d'un état (initial ou fixé), et dans un environnement donné, le circuit, après exécution pendant un nombre de cycles (de simulation, temporels, d'horloge, etc...) fixé, se trouve dans l'état final (dans lequel le résultat est disponible) et que certaines valeurs de la sortie et/ou de l'état correspondent à la valeur de la spécification pour les mêmes valeurs d'entrée.

Dans la description ci-dessus, apparaissent quelques éléments qui ont besoin d'une définition formelle tels que : l'environnement, l'état final, la valeur de la spécification pour les mêmes valeurs d'entrée. En effet, leurs définitions correspondent à deux types d'abstraction :

1) La définition de l'environnement correspond à une *abstraction comportementale* du circuit. Le circuit VHDL a généralement des comportements différents en fonction de ses entrées. Ainsi, il existe un environnement attendu pour son bon fonctionnement. Le comportement du circuit dans un tel environnement représente une abstraction comportementale par rapport au circuit initial. La définition d'un bon environnement est un prérequis pour énoncer le théorème de correction. Soit $Constrained(Input)$ l'abstraction comportementale du $System$ et $Final(st)$ la définition de l'état final du circuit.

2) Pour définir la valeur de la spécification pour les mêmes entrées du circuit, il faut définir une *abstraction de données* pour le circuit. Le circuit travaille probablement sur un nombre beaucoup plus grand d'éléments que la spécification, et il n'y a pas de relation biunivoque entre les éléments de la spécification et ceux du circuit. Les deux modèles diffèrent aussi dans le type de leurs données. D'habitude, l'implémentation travaille sur des entiers bornés, ou des vecteurs de bits, tandis que la spécification utilise des entiers. Il faut donc définir une fonction qui lie les entrées du circuit avec celles de la spécification. Soit $Abstraction(Input, params)$ le prédicat qui établit le lien entre l'entrée du circuit et les bons paramètres pour la spécification $Spec$. La même relation doit être définie entre la sortie du circuit et le résultat de la spécification. Soit $Result(st, Spec(params))$ le prédicat qui définit ce lien.

Formellement, le *théorème de correction* du circuit qu'on veut prouver a la forme suivante :

$$\left.\begin{array}{l} \forall Input.\ Constrained(Input)\ \wedge \\ \forall params.\ Abstraction(Input, params)\ \wedge \\ \forall st \end{array}\right\} \Rightarrow \begin{array}{l} Final(st_{new})\ \wedge \\ Result(st_{new}, Spec(params)) \end{array}$$

où $st_{new} = System(steps(Input), st)$

La fonction *steps* renvoie la liste de suites d'entrées nécessaires pour que le circuit finisse le calcul. Dans les cas simples, la longueur de la liste renvoyée par la fonction *step* est une constante, mais en général elle dépend des entrées du circuit. Pour les circuits synchronisés par horloge, la longueur de la liste correspond au nombre de cycles d'horloge pour lequel le circuit est exécuté.

Maintenant que nous avons défini formellement le théorème de correction, il reste à le prouver. Une manière de réaliser la preuve est d'exécuter symboliquement la fonction

System pendant le nombre de pas définis pas *steps*, et ensuite prouver que le résultat du calcul symbolique vérifie la propriété. Ceci peut être efficace si le nombre de pas est petit, et que le système n'est pas très grand. Même dans ces cas, la preuve du théorème n'est pas automatique, car elle est très dépendante de la définition de la spécification.

Par contre, si le nombre de pas donnés par *step* est grand ou/et le système a beaucoup d'éléments, vu que la taille de l'expression calculée par *System* croît de manière exponentielle, le démonstrateur arrive vite à ses limites. Le temps de réponse de l'outil croît aussi exponentiellement avec le nombre de pas, car à chaque pas l'outil applique un grand nombre de règles de simplification. Dans ce cas, il est conseillé d'adopter une approche compositionnelle. Nous avons montré dans le chapitre précédent que la fonction *System* (la traduction d'un circuit en ACL2), telle que nous la définissons, a la propriété suivante (où & est l'opérateur de concaténation sur des listes) :

Sys : $\forall Input_1, Input_2, st.$

$$System(Input_1, System(Input_2, st) = System(Input_1 \& Input_2, st)$$

Grâce à cette propriété, le comportement du circuit peut être décomposé et le raisonnement est porté sur un nombre plus petit de pas.

Nous avons trouvé que le critère de décomposition est étroitement lié aux étapes de calcul définies par le concepteur dans l'automate de contrôle de la description VHDL.

Nous présentons l'automate de contrôle d'une description VHDL par un diagramme d'état dans le style UML, en utilisant les trois éléments suivants :

- *l'etat* dans le diagramme représente une étape de calcul dans lequel le circuit peut rester pendant plusieurs cycles. Dans une étape de calcul, certaines des valeurs des objets du circuit peuvent être modifiées.
- la *transition* d'une étape de calcul à l'autre est réalisée si une condition de transit *Cond* est remplie.
- le *point de contrôle* d'une étape de calcul représente une configuration des valeurs d'objets du circuit avec laquelle l'étape de calcul débute.

Nous avons deux types de parcours (Figure 4.2) : soit une succession simple d'étapes de calcul (a), soit une boucle (b).

Dans le premier cas, pour prouver l'étape de calcul S_0, il faut montrer le théorème : si le système se trouve au point de contrôle a, après n exécutions dans le bon environnement, le système se trouve au point de contrôle b, où n est soit une constante, soit il dépend de l'état courant du système et des entrées. Le théorème s'écrit dans la logique de la manière suivante ($Firstn(k, L)$ renvoie les premiers k éléments de la liste L) :

Etape-S_0 :

$$\left.\begin{array}{l} \forall Input.\ Constrained_a(Input)\ \wedge \\ \forall st.\ a(st) \end{array}\right\} \Rightarrow\ b(System(Firstn(n, Input), st))$$

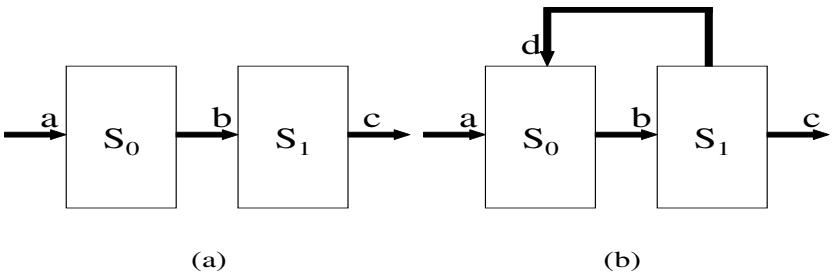

FIG. 4.1 – Types de successions d'étapes de calcul dans un automate de contrôle VHDL : a) simple ; b) une boucle.

Si n est suffisamment petit, le théorème précédent peut se prouver par exécution symbolique. Sinon, la preuve peut être réalisée par récurrence. Pour cela, il faut prouver une forme généralisée du théorème Etape-S_0. Tout d'abord, il faut définir un invariant par rapport à l'étape de calcul sur l'état st et un sur les entrées du circuit. Soit $Inv(st)$ l'invariant de l'état dans l'étape S_0, et $Constrained_{Inv}(Input)$ l'invariant sur les entrées. La forme généralisée du théorème garantit qu'à partir d'un état quelconque de l'étape de calcul ($Inv(st)$), toutes les exécutions du système, jusqu'à celle qui fait changer l'étape, gardent l'invariant. C'est à dire que le système reste dans la même étape de calcul.

Etape-S_0-gen :

$$\left.\begin{array}{l} \forall st.\ Inv(st)\ \wedge \\ \forall Input.\ Constrained_{Inv}(Input)\ \wedge \\ \forall k.\ 0 \leq k \leq n \end{array}\right\} \Rightarrow \begin{array}{l} \textbf{if}\ Cond\ \textbf{then} \\ \quad b(System(Firstn(k, Input), st)) \\ \textbf{else}\ Inv(System(Firstn(k, Input), st)) \end{array}$$

Pour prouver ce théorème, il faut utiliser le schéma de récurrence suivant (T est le théorème à prouver) :

1. Cas de base : $T(0, st, Input)$.
2. Pas de récurrence : $T(k{-}1, System(First(Input), st), Rest(Input)) \Rightarrow T(k, st, Input)$.

Ainsi que les lemmes suivants :

1. L'invariant sur l'état est préservé : si le système est dans un état st qui respecte l'invariant Inv et si la condition de sortie de l'étape de calcul, $Cond$, n'est pas remplie, alors, après une exécution, l'état du système garde l'invariant. Si par contre, la condition de sortie de l'étape de calcul, $Cond$, est remplie, alors après une exécution le système se

trouve dans un état qui respecte le prédicat b.

$$\left.\begin{array}{l}\forall st.Inv(st) \wedge \\ \forall Input.Constrained_{Inv}(Input)\end{array}\right\} \Rightarrow \begin{array}{l}\textbf{if } Cond \textbf{ then} \\ \quad b(System(First(Input), st)) \\ \textbf{else } Inv(System(First(Input), st))\end{array}$$

2. L'invariant sur les entrées est préservé.

$$\forall Input.Constrained_{Inv}(Input) \Rightarrow Constrained_{Inv}(Rest(Input))$$

Maintenant, pour prouver le théorème Etape-S_0, il est suffisant de :

- prouver que le prédicat a garde l'invariant :

$$a(st) \Rightarrow Inv(st)$$

- prouver que les contraintes pour les entrées $Constrained_a$ gardent l'invariant :

$$Constrained_a(Input) \Rightarrow Constrained_{Inv}(Input)$$

- instancier le théorème généralisé pour $k = n$.

Si l'on veut prouver le comportement du circuit pour les étapes de calcul S_0 et S_1 (Figure 5.1, a), il faut prouver le théorème : à partir du point de contrôle défini par a, après l'exécution du système pour m pas dans le bon environnement, le système se trouve dans le point de contrôle défini par c.

Etape-S_0-et-S_1 :

$$\left.\begin{array}{l}\forall Input.\ Constrained_c(Input)\ \wedge \\ \forall st.\ a(st)\end{array}\right\} \Rightarrow\ c(System(Firstn(m, Input), st))$$

Pour prouver le théorème Etape-S_0-et-S_1 il est suffisant de prouver :

- le théorème pour l'étape de calcul S_0, Etape-S_0
- le théorème similaire pour S_1 : à partir du point de contrôle b, après p pas, le système se trouve dans le point de contrôle c.

  Etape-S_1 :

$$\left.\begin{array}{l}\forall Input.\ Constrained_b(Input)\ \wedge \\ \forall st.\ b(st)\end{array}\right\} \Rightarrow\ c(System(Firstn(m-n, Input), st))$$

- combiner les deux théorèmes avec le théorème Sys.

Traitons maintenant le cas d'une boucle dans le comportement du circuit (Figure 5.1, b). Nous sommes intéressés par la vérification du comportement du circuit à partir du point de contrôle a jusqu'au point de contrôle c. Ce cas est similaire au cas précédent. En effet, si nous considérons que ce cas est un raffinement de l'étape S_0 du cas précédent, alors un pas de l'étape S_0 précédente est devenu une itération de la boucle dans le nouveau système. Nous allons quand même détailler cette configuration afin d'offrir un modèle de

raisonnement complet.

Le comportement du circuit de point de contrôle a jusqu'au b est le même que dans le cas précédent. Ce qui change est le comportement du circuit à partir du point de contrôle b jusqu'au c. Puisque le circuit boucle entre les étapes de calcul S_0 et S_1, le circuit va aller du point b à d en i cycles, ensuite de d à b en j cycles, et répéter la séquence r fois, jusqu'à ce que la condition $Cond$ qui déclenche l'étape qui débute avec c soit remplie (en effet cette condition correspond à la condition d'arrêt de la boucle) et dans ce cas il va aller de b à c en p pas. Donc pour que le circuit arrive du point a en c il exécute $n + (i + j) * r + p$ pas. Pour prouver cette propriété, le comportement du circuit est d'abord décomposé comme présenté ci-dessus, et pour chaque transition d'un point à l'autre un théorème est prouvé. Pour les points b et d, les prédicats correspondants qui décrivent l'état sont en effet des invariants de la boucle.

Nous continuons avec l'évaluation du comportement du circuit pour une itération de la boucle en combinant le comportement du circuit du point b à d et de d à b. Nous obtenons ainsi un théorème qui décrit le comportement du circuit du point b au prochain point b de la boucle. Ce théorème donne l'invariant de la boucle.

Maintenant nous prouvons par récurrence le comportement du circuit pour la boucle dans sa totalité : à partir du premier point de contrôle b, après l'exécution du système pendant $(i + j) * r + p$ cycles, le système se trouve au point de contrôle c.

Etape-S_1 :

$$\left.\begin{array}{l} \forall Input.\ Constrained_b(Input)\ \wedge \\ \forall st.\ b_{init}(st) \end{array}\right\} \Rightarrow\ c(System(Firstn((i+j)*r+p, Input), st))$$

Pour prouver ce théorème il faut avant tout, comme dans le cas précédent, prouver une forme plus générale : à partir d'un point b quelconque de la boucle, si la condition de sortie de la boucle est remplie, alors le système arrive au point c, sinon il reste dans la boucle.

Etape-S_1-gen :

$$\left.\begin{array}{l} \forall Input.\ Constrained_b(Input)\ \wedge \\ \forall st.\ b(st)\ \wedge \\ \forall k.\ 0 \leq k \leq r \end{array}\right\} \Rightarrow$$

$$\begin{array}{l} \textbf{if}\ Cond\ \textbf{then} \\ \quad c(System(Firstn(k*(i+j)+p, Input), st)) \\ \quad \textbf{else}\ b(System(Firstn(k*(i+j), Input), st)) \end{array}$$

Pour prouver ce théorème, il faut utiliser un schéma de récurrence similaire au schéma défini dans le cas précédent. La différence est que le saut est réalisé pour le nombre de pas, $i + j$, nécessaire pour réaliser une itération de la boucle, et non seulement 1 pas comme dans la définition précédente (T est le théorème à prouver) :

1. Cas de base : $T(0, st, Input)$.

2. Pas de récurrence :
$T(k-1, System(Firstn(i+j, Input), st), Restn(i+j, Input)) \Rightarrow T(k, st, Input)$.

Les lemmes auxiliaires sont similaires aussi :
1. L'invariant sur la boucle est préservé :

$$\left.\begin{array}{l} \forall st.b(st) \wedge \\ \forall Input.Constrained_b(Input) \end{array}\right\} \Rightarrow$$

$$\begin{array}{l} \textbf{if } Cond \textbf{ then} \\ \quad c(System(Firstn((i+j)+p, Input), st)) \\ \quad \textbf{else } b(System(Firstn((i+j), Input), st)) \end{array}$$

2. L'invariant sur les entrées est préservé.

$$\forall Input.Constrained_b(Input) \Rightarrow Constrained_b(Restn(i+j, Input))$$

Pour obtenir le théorème général sur le comportement du circuit de a à b, il suffit de composer les preuves avec le théorème Sys.

Tout automate de contrôle d'un circuit peut se ramener à une combinaison de ces deux types de succession d'étape de calcul. Ainsi, en suivant la méthodologie de preuve présentée, le théorème général sur le comportement entier du circuit est obtenu.

Notre méthode est basée sur des points de contrôle définis sur le comportement du circuit. La définition de ces points revient entièrement à l'utilisateur. Dans la plupart des cas, vu la différence de comportement et de données entre la spécification et l'implémentation, il est très difficile de définir le point de contrôle en étroite relation avec la spécification. Dans ce cas il est conseillé d'utiliser des fonctions intermédiaires, qui ont la même construction que la spécification, mais qui ont comme paramètres des éléments de l'implémentation. Ainsi l'effort de preuve est diminué, même si la quantité de preuve augmente.

Dans ce cas la preuve a lieu en deux temps :

- dans un premier temps nous prouvons le comportement du circuit entre les points de contrôle définis :

$$\left.\begin{array}{l} \forall Input.\ Constrained(Input) \ \wedge \\ \forall st \end{array}\right\} \Rightarrow \begin{array}{l} Final(System(steps(Input), st)) \ \wedge \\ Result_{implem}(System(steps(Input), st)) \end{array}$$

- dans un deuxième temps nous prouvons l'équivalence, sous l'abstraction de données, entre les fonctions intermédiaires et la spécification.

$$\forall params_{implem}, params_{spec}.\ Abstraction(Input, params_{spec}) \Rightarrow$$

$$Result_{implem}(System(steps(Input), st)) = Spec(params_{spec})$$

En combinant les deux théorèmes, nous obtenons le théorème de correction du circuit.

Nous avons appliqué cette approche de vérification pour deux types de composants cryptographiques : l'un implémentant l'algorithme de hachage SHA-1, et l'autre implémentant l'algorithme de cryptage Triple DES.

Par la suite, nous discutons les deux schémas cryptographiques, leurs modélisations dans la logique de premier ordre et la vérification des modules matériels les implémentant.

4.3 Les fonctions de hachage

Les fonctions de hachage cryptographiques sont une des primitives fondamentales dans la cryptographie moderne. Elles sont utilisées avec les algorithmes de clé publique pour le cryptage et la signature digitale. Elles sont aussi utilisées dans la vérification de l'intégrité du message et dans l'authentification.

De manière générale, une fonction de hachage traite un message de longueur variable afin de produire un message condensé.

A première vue, les fonctions de hachage cryptographiques semblent les mêmes que les fonctions de hachage classiques utilisées dans les applications informatiques autres que la cryptographie. Dans les deux cas, elles prennent en entrée un message et produisent en sortie un message condensé, donc des domaines plus grands sont confondus dans des domaines plus petits. Ceci fait que plusieurs éléments du domaine d'entrée doivent correspondre au même élément du domaine de sortie (phénomène appelé collision).

Les fonctions de hachage cryptographiques doivent en plus remplir les deux conditions suivantes :

- il est facile de calculer le condensé d'un message, mais il est impossible en pratique (c'est à dire qu'il n'est pas possible de le réaliser dans un temps raisonnable) de recréer le message initial à partir d'un message condensé.
- les collisions doivent être difficiles à trouver du point de vue du calcul, donc en pratique elles ne doivent jamais arriver. Une fonction de hachage bien construite, avec une sortie de n bits, a une probabilité de $2^{n/2}$ qu'un message arbitraire soit associé avec un message condensé donné.

Il existe de nombreuses fonctions de hachage proposées dans la littérature. La plupart d'entre elles ont déjà subi des attaques qui montrent leur fragilité. Réussir une attaque sur une fonction de hachage équivaut à montrer qu'au moins une des propriétés mentionnées n'est pas vraie. Une des premières fonctions de hachage très utilisées est MD5 [111]. Elle a été définie par Ron Rivest en 1992, comme une amélioration de son ancienne proposition de 1990, MD4 [112]. En 1993, L'Agence Nationale de Sécurité des Etats Unis (NSA) publie une fonction de hachage similaire à MD5, appelée SHA[1] [95]. En 1995, à cause d'une faiblesse de l'algorithme, le NSA change l'algorithme, la nouvelle version étant SHA-1 [96]. Aujourd'hui la fonction de hachage la plus populaire est SHA-1, tandis que MD5 est toujours utilisée dans de nombreuses applications.

Une première attaque sur des versions simplifiées de la fonction MD5 a été annoncée en 1996 [41], mais en 2005 ont été enregistrées des attaques sur des versions complètes de l'algorithme [140, 64]. De même pour la première version de SHA [135]. Pour la deuxième version de SHA, SHA-1, des attaques ont aussi été rapportées [134], mais pour l'instant

[1] Secure Hash Algorithm

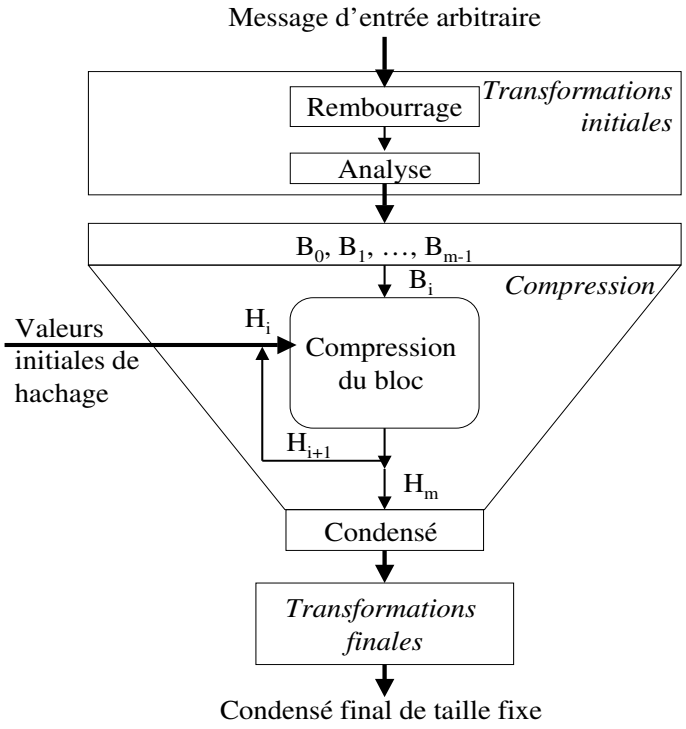

FIG. 4.2 – Modèle général d'une fonction de hachage

elles sont bénignes, dans le sens où une faiblesse théorique a été mise en évidence, mais pas des collisions. Pour contrecarrer la montée en puissance de calcul des ordinateurs, des versions plus résistantes aux attaques de SHA ont été définies : ce sont les variantes SHA-224, SHA-256, SHA-384 et SHA-512, connues aussi sous le nom de la famille SHA-2 [97].

Le principe des fonctions de hachage est illustré par la figure 5.2. L'algorithme prend en entrée un message M, et une valeur initiale de hachage H, et retourne un message condensé, de la même taille que la valeur de hachage.

Le message d'entrée M, qui est une séquence de bits de longueur arbitraire, est d'abord soumis à un processus de transformation à deux étapes (n est un paramètre de l'algorithme) :

- **Rembourrage** : M est complété avec des bits jusqu'à avoir une longueur multiple de n. Soit *Padding* la fonction qui implémente cette opération, et $n * m$ la longueur de son résultat.
- **Analyse** : le message rembourré est coupé en m blocs de n bits. Soit *Parsing* la fonction qui réalise cette opération.

Ensuite, pour chaque bloc B_i du message et la valeur de hachage H_i, un algorithme

itératif calcule le message condensé du bloc. Soit *Digest-one-block* la fonction qui implémente l'algorithme de compression pour un bloc. Le message condensé obtenu est considéré comme valeur de hachage H_{i+1} pour le prochain bloc B_{i+1}. Le condensé du dernier bloc, B_m, est potentiellement transformé encore une fois par une suite d'opérations implémentées par $Update_{final}$. Le résultat du calcul est le condensé du message entier. Le processus de compression débute avec une valeur initiale de hachage H qui généralement est une constante de l'algorithme.

Un modèle fonctionnel général de la fonction de hachage est réalisé par la fonction $Hash$:

$$Hash\ (\mathrm{M},\ \mathrm{H}) \stackrel{def}{=} Update_{final}(\mathrm{H},\ Digest(Parsing(Padding(\mathrm{M})),\ \mathrm{H}))$$

Digest calcule le condensé pour les blocs du message. Si tous les blocs ont été compressés, alors la valeur retournée est la valeur de hachage finale, sinon le premier bloc est compressé et la valeur de son condensé est considérée comme valeur de hachage pour le calcul du condensé des blocs restants.

$Digest(List_{blocks},\ \mathrm{H}) \stackrel{def}{=}$
if $Empty(List_{blocks})$ **then** H
else let* $\mathrm{B} = First(List_{blocks})$
$Suite_{blocks} = Rest(List_{blocks})$
$\mathrm{H}_i =$ *Digest-one-block*(B,H)
in
$Digest(Suite_{blocks},\ \mathrm{H}_i)$
fi

La compression d'un bloc est un algorithme itératif illustré en Figure 4.3. Au préalable sont effectuées des opérations d'initialisation du calcul pour le bloc et pour la valeur de hachage. Ensuite, des opérations sont appliquées de manière itérative sur le bloc un nombre K de fois (K est un paramètre de l'algorithme). Comme résultat, tant le bloc que le condensé sont modifiés. Après le calcul de la boucle, le condensé est encore soumis à des transformations afin d'obtenir le résultat final.

Le modèle fonctionnel de la compression d'un bloc est :

Digest-one-block(B,H) $\stackrel{def}{=} Update(\mathrm{H},$ *Digest-step*$(0,\ \mathrm{K},\ Init_{block}(\mathrm{B},\mathrm{H}),\ Init_{hash}(\mathrm{B},\mathrm{H})))$

Digest-step(j, K, B, H) $\stackrel{def}{=}$
if $\mathrm{K} \leq \mathrm{j}$ **then** H
else let $\mathrm{B}_{j+1} = Operations_{block}(\mathrm{j},\ \mathrm{B},\ \mathrm{H})$
$\mathrm{H}_{j+1} = Operations_{hash}(\mathrm{j},\ \mathrm{B},\ \mathrm{H})$
in
Digest-step(j+1, K, B_{j+1}, H_{j+1})
fi

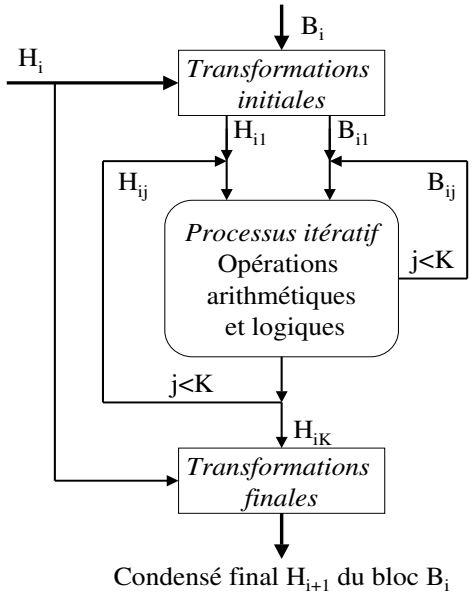

FIG. 4.3 – Modèle général pour la compression d'un bloc

Notre modèle de fonction de hachage est un modèle général. Dans les cas concrets des algorithmes de hachage, certaines des étapes mises en évidence ici n'existent pas. Par exemple, l'algorithme SHA-1 n'a pas de transformations finales pour le condensé d'un message. Dans ce cas, la fonction $Update_{final}$ ne modifie pas le condensé du dernier bloc.

Le reste de la section s'organise de la manière suivante : tout d'abord nous présentons l'algorithme SHA-1 et sa modélisation comme instance du modèle fonctionnel général. Ensuite nous présentons les caractéristiques du circuit qui implémente l'algorithme. Finalement nous montrons comment appliquer l'approche de vérification présentée en début du chapitre pour prouver que la description matérielle implémente correctement la spécification.

4.3.1 Spécification de l'algorithme SHA-1

Le SHA (Secure Hash Algorithm) est une famille standardisée d'algorithmes de hachage. Les cinq versions de SHA : SHA-1, SHA-224, SHA-256, SHA-384 et SHA-512, diffèrent par la dimension des blocs, des mots, de la longueur du message condensé, mais aussi par la manière d'obtenir le condensé pour un bloc. Un comparatif des paramètres pour les fonctions est présenté dans le tableau 4.1, où la colonne *Message* donne la longueur maximale du message pour lequel l'algorithme respecte les propriétés d'une fonction de hachage. La colonne suivante, *Bloc*, donne la longueur d'un bloc sur lequel est appliquée l'étape de compression. *Mot* donne la longueur des mots sur lesquels les opérations logiques

Algorithme	Message (bits)	Bloc (bits)	Mot (bits)	Condensé (bits)
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512

TAB. 4.1 – Les variantes de SHA

et arithmétiques sont appliquées. La colonne *Condensé* donne la longueur du condensé pour chaque algorithme.

Pour le SHA-1, le message d'entrée M, qui est une séquence de bits de longueur arbitraire L$< 2^{64}$, est modifié comme suit :

- **Rembourrage** : M est concaténé avec un bit à 1, suivi de k bits à 0, suivi de la représentation sur 64 bits de L. k est la plus petite solution naturelle de l'équation : (L+1+k) mod 512 = 448. Le message après rembourrage a une longueur multiple de 512.
- **Analyse** : le message rembourré est coupé en blocs de 512 bits.

Ensuite, les blocs résultants sont traités (conformément à la fonction *Digest*). Le resultat obtenu pour le dernier bloc est le condensé du message. Ainsi, l'algorithme n'a pas de transfomation finale pour le condensé : $Update_{final}$(H, *digest*)=*digest*.

L'algorithme qui calcule le condensé d'un bloc a 80 itérations. Un bloc est vu comme une séquence de 16 mots de 32 bits. Durant l'algorithme, les mots d'un bloc sont combinés avec les contenus de cinq registres internes (A, B, C, D, E), de 32 bits chacun, en utilisant des opérations logiques et arithmétiques. Au début du calcul, les registres internes sont initialisés avec des constantes prédéfinies. Après l'application de l'algorithme pour un bloc, les registres internes sont additionnés à la valeur de hachage du bloc, afin d'obtenir le message condensé du bloc.

Conformément au standard SHA-1, l'étape de calcul du message condensé utilise le bloc de 512 bits coupé en 16 mots W_i $(0 \leq i \leq 15)$ afin de générer 80 mots. Les 16 premiers mots générés sont les mots du bloc initial. Les 64 mots restants sont obtenus en appliquant des opérations logiques XOR et shift sur des mots déjà existants, conformément à l'algorithme suivant, où S^n indique une opération shift de n bits à gauche, et '+' est l'addition entre deux vecteurs de bits.

for t=16 to 79

W_t=S^1 (W_{t-3} XOR W_{t-8} XOR W_{t-14} XOR W_{t-16}) endfor.

Les cinq variables A, B, C, D, E sont calculées d'après l'instruction :

for t=0 to 79 do

TEMP=$S^5(A)$ + f_t (B, C, D) + E + W_t + K_t ;

E=D ; D=C ; C=S^{30}(B) ; B=A ; A=TEMP ;
endfor.

où f_t sont des fonctions et K_t sont des constantes définies dans le standard. Avant de commencer l'exécution de la boucle, les variables A, B, C, D, E sont initialisées avec les valeurs constantes H_0, H_1, H_2, H_3, H_4 spécifiées dans le standard.

Quand le message condensé d'un bloc a été obtenu, les valeurs H_i sont mises à jour :
$H_0 = H_0$+A ; $H_1 = H_1$+B ; $H_2 = H_2$+C ; $H_3 = H_3$+D ; $H_4 = H_4$+E ;

Le message condensé du message initial est la concaténation des valeurs H_0, H_1, H_2, H_3, H_4 obtenue pour le dernier bloc à traiter.

Le standard propose un algorithme alternatif pour le calcul de SHA-1, moins coûteux en espace mémoire. Pour mémoriser seulement 16 mots, au lieu de 80, les mots W_t et les variables A, B, C, D, E sont calculés dans la même boucle, et chaque mot W_t, où $16 \leq t$ est écrit sur $W_{t \bmod 16}$. Dans l'implémentation que nous avons étudiée, cette variante a été choisie, afin d'optimiser la dimension du circuit. Nous présentons le principe de l'algorithme alternatif, sachant que nous avons prouvé à l'aide d'ACL2 que les deux variantes de l'algorithme sont équivalentes.

for j=0 to 79 do
  s=j$\wedge$MASK ;
  if j$\geq$16 then
    W_s=$ROTL^1$ ($W_{(s+13)\wedge MASK}$ XOR $W_{(s+8)\wedge MASK}$
      XOR $W_{(s+2)\wedge MASK}$ XOR W_s) ;
  endif ;
  TEMP=$ROTL^5(A)$ + F_j (B, C, D) + E + W_s + K_j ;
  E=D ; D=C ; C=$ROTL^{30}$(B) ; B=A ; A=TEMP ;
endfor.

où MASK = '00000000000000000000000000001111' est une constante du standard.

Exemple : Soit l'application de l'algorithme SHA-1 au message "abc". La longueur du message est $8 \times 3 = 24$. Le message rembourré est : "abc", suivi d'un bit à 1, et $448-(24+1) = 423$ bits à 0, et ensuite la longueur du message. Le résultat est un message de 512 bits : :

$$\underbrace{01100001}_{a}\ \underbrace{01100010}_{b}\ \underbrace{01100011}_{c}\ 1\ \overbrace{00..00}^{423}\ \overbrace{00..011000}^{64}$$

Le message condensé (en hexadécimal) pour "abc" est :
A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D.

Ainsi, en suivant le schéma de définition du modèle fonctionnel d'une fonction de hachage, la fonction *Digest-one-block* a la forme :

Soit ABC_Vars $\stackrel{def}{=}$ *List* (A, B, C, D, E) la liste des variables A,B, C, D, E.

Digest-one-block(Block, ABC_Vars) $\stackrel{def}{=}$
Update(ABC_Vars,
Digest-step(0, 80,
$Init_{block}$(Block,ABC_Vars),
$Init_{hash}$(Block,ABC_Vars)))

Mais l'algorithme SHA-1 n'a pas d'opération d'initialisation, donc :

$Init_{block}$(Block, ABC_Vars) = Block, et
$Init_{hash}$(Block, ABC_Vars)= ABC_Vars

La fonction *Update* ajoute les valeurs retournées par le processus itératif à la valeur de hachage pour le bloc (+ est l'addition de deux vecteurs de bits) :

Update(H,*digest*)= H + *digest*

Maintenant il reste à définir la fonction *Digest-step* qui, conformément au modèle général, a la forme :

Digest-step (j, 80, Block, ABC_Vars) $\stackrel{def}{=}$
if *Natural* (j) **then**
if $80 \leq j$ **then** ABC_Vars
else let New-Block **be**
$Operations_{block}$(j, Block, ABC_Vars)
New-ABC_Vars **be**
$Operations_{hash}$(j, New-Block, ABC_Vars)
in
Digest-step (j+1, New-Block, New-ABC_Vars)
fi
else *nil*
fi

Pour compléter le modèle il faut définir les fonctions $Operations_{block}$ et $Operations_{hash}$.

La modification du bloc pendant un pas de l'algorithme de compression de SHA-1, est modélisée de la manière suivante :

$Operations_{block}$(j, Block, ABC_Vars) $\stackrel{def}{=}$
if $16 \leq j$ **then** *Replace* (s (j), *Word_Spec* (j, Block), Block)
else Block **fi**

La fonction *Replace* (i, W, Block) remplace la valeur du mot i de Block avec W. La fonction *Word_Spec* calcule le mot $W_{s(j)}$ du pas j de l'algorithme, où s (j) calcule j *mod* 16. Voici la définition de *Word_Spec*, conformément à l'algorithme :

Word_Spec (j, Block) $\stackrel{def}{=}$
Rotl-spec (1, *B-Xor* (Block[*B-And* (*mask*, 13 + s (j))],
Block[*B-And* (*mask*, 8 + s (j))],

Block[*B-And* (*mask*, 2 + s (j))],
Block[*B-And* (*mask*, j)]))

B-And et *B-Xor* sont des macros qui calculent les opérations logiques 'et' et 'ou exclusif' entre deux bits, et deux vecteurs de bits de longueurs éventuellement différentes. L'opération + est surchargée avec l'addition entre deux vecteurs de bit, et entre un naturel et un vecteur de bit, *Rotl-spec (n, w)* est le déplacement circulaire de n bits du vecteur de bit w.

La modification des registres A, B, C, D, E, qui stockent la valeur de hachage, est modélisée par la fonction $Operations_{hash}$.

$Operations_{hash}$(j, New-Block, ABC_Vars) $\stackrel{def}{=}$
⟨*Temp_Spec* (j, ABC_Vars, New-Block), A, *Rotl* (30, B), C, D⟩

La fonction *Temp_Spec* calcule la variable TEMP pour le pas j.

Temp_Spec (j, ABC_Vars, Block) $\stackrel{def}{=}$
Rotl-spec (5,A) + *F-spec* (j, B, C, D) + E + Block[s(j)] + K(j)

s (j) $\stackrel{def}{=}$ *Bv-nat-be* (*B-And* (*Nat-bv-be* (j), *mask*))

Bv-nat-be, *Nat-bv-be* sont des fonctions de conversion conformes à la représentation "big endian" et *F-spec* modélise les fonctions F_j de l'algorithme.

Sha_ Spec définit la spécification pour SHA-1 :

Sha_Spec (Message, Initial_Hash_Val) $\stackrel{def}{=}$
Digest (*Parsing* (*Padding* (Message), Initial_Hash_Val)

Un message arbitraire est d'abord soumis à l'étape de rembourrage et ensuite coupé en blocs de 512 bits. La liste de blocs est traitée avec les valeurs initiales constantes de hachage : Initial_Hash_Val $\stackrel{def}{=}$ ⟨*h0*, *h1*, *h2*, *h3*, *h4*⟩.

Validation de la spécification

L'algorithme de SHA-1 est formalisé en ACL2 [124], et il a été exécuté sur les tests fournis avec la documentation standard. Une validation complémentaire est obtenue par la vérification des propriétés mathématiques et de sûreté de l'algorithme en utilisant le démonstrateur ACL2. En effet, nous avons aussi défini les cinq versions de SHA : SHA-1, SHA-244, SHA-256, SHA-384 et SHA-512 comme des instances du modèle général. Environ 70 définitions de fonctions et environ 100 lemmes sur les fonctions ont été écrites. Parmi les propriétés de sûreté prouvées pour SHA-1 :

- La longueur du message après rembourrage est un multiple de 512, strictement plus grand que 0.
- Les 64 derniers bits du message après rembourrage représentent le code binaire de la longueur du message initial.
- Les L premiers bits du message après rembourrage représentent le message initial.

– Les bits entre le bit de fin du message et les 64 derniers bits sont tous 0.
– Après l'analyse, le résultat est une liste de blocs, de 512 bits chacun.
– Le résultat final de SHA-1 est un message condensé sur 160 bits.

A cause de sa nature, il n'est pas facile d'écrire une expression mathématique représentant le calcul pour le message condensé. Donc, le code Lisp de l'algorithme itératif est la définition du calcul.

4.3.2 Caractéristiques du circuit et sa modélisation en ACL2

Le circuit SHA-1 a été fourni par nos partenaires du projet ISIA-2. La description n'implémente pas tout l'algorithme, mais considère que l'entrée du circuit est le message après l'étape de rembourrage. Le circuit prend des mots de 32 bits en entrée, à partir d'une mémoire RAM externe. La mémoire est aussi utilisée pour stocker les mots W_s calculés durant l'algorithme.

Le modèle est écrit dans le sous-ensemble de VHDL synthétisable et il est synchrone. Les entrées et les sorties du circuit sont décrites dans le tableau 4.2. Le circuit a aussi 23 signaux internes. Nous allons indiquer seulement une partie d'entre eux, qui semble importante pour illustrer notre approche : a, b, c, d, e sont les registres de 32 bits qui gardent le message condensé, *state* est un vecteur de 3 bits qui donne l'état de l'automate de contrôle, $blocks\_left$ est un vecteur de 6 bits qui représente le nombre de blocs qu'il reste à traiter, *count* est un mot de 8 bits qui compte les itérations de l'algorithme.

start, reset	input	bit	début du calcul, reset asynchrone
reset_done	input	bit	remet à 0 la sortie *done*
clk	input	bit	l'horloge
rdata	input	32-bit	le mot d'entrée du message
base_address	input	12-bit	l'adresse RAM où le message est gardé
nb_block	input	6-bit	nombre de blocs
address	output	12-bit	l'adresse courante de RAM
ram_sel,	output	bit	signal de sélection de la RAM
ram_write	output	bit	signal d'écriture sur la RAM
wdata	output	32-bit	nouveau W à écrire dans la RAM
busy	output	bit	signale que le circuit calcule
done	output	bit	le message condensé est obtenu
aout, bout, cout, dout, eout	output	32-bit	le message condensé

TAB. 4.2 – Les entrées et les sorties du circuit implémentant SHA-1

Le circuit SHA-1 est composé d'une partie contrôle et une partie donnée. La Figure 4.4 montre le graphe de transition pour l'automate de contrôle VHDL.
Le comportement global du circuit est le suivant :

– *idle* est l'état d'attente ;
– dans l'étape *init* le circuit écrit les valeurs initiales constantes de hachage H_0 to H_4 dans les registres A, B, C, D, E ;

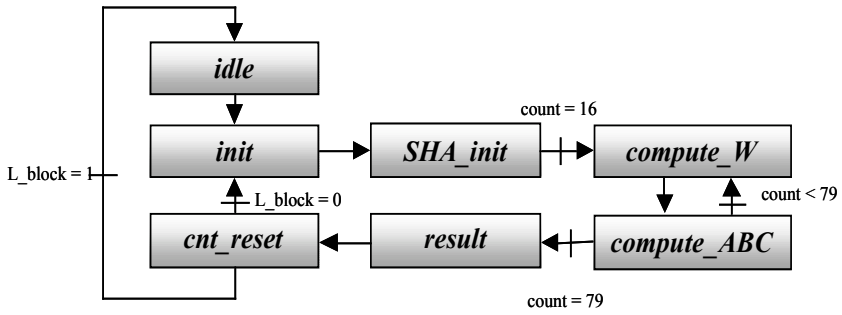

FIG. 4.4 – Automate de contrôle du circuit SHA-1

- dans l'étape *SHA_init* le circuit lit un bloc à partir de la RAM par mots de 32 bits et calcule les 16 premières valeurs pour A, B, C, D, E. Dans d'autres mots, le circuit exécute les 16 premières itérations de l'algorithme ;
- dans l'étape *compute_W* le circuit calcule un des 64 mots, W ;
- dans l'étape *compute_ABC* le circuit calcule les valeurs de A, B, C, D, E correspondant au mot calculé précédemment, W, et met à jour sa valeur dans la RAM.
  Les états *compute_W* et *compute_ABC* sont répétés 64 fois ;
- *result* ajoute les dernières valeurs de A, B, C, D, E aux dernières valeurs de H_0 to H_4 ;
- dans l'étape *cnt_reset* le circuit initialise les différents compteurs ; si le dernier bloc a été traité, (L_block = 1) alors le signal *done* est mis à 1, indiquant que le résultat est disponible.

La mémoire RAM a le comportement suivant :

- si le bit *ram_sel* est à 1, alors les opérations sur la RAM sont permises ;
- l'écriture est permise seulement si le bit *ram_write* est à 1, et dans ce cas la donnée *wdata* est écrite à l'*address* ;
- si le bit *ram_write* est à 0, la lecture est permise, et la donnée de l'*address* est mise dans *rdata*, qui est une entrée du circuit.

Formalisation du circuit SHA-1 dans ACL2

La description VHDL est traduite automatiquement dans un modèle fonctionnel en utilisant la méthode décrite dans le chapitre 3. Le modèle est simulé symboliquement pour un cycle d'horloge. Ensuite, le système d'équations récurrentes résultant est traduit en ACL2, comme décrit dans le chapitre 4.

Donc un pas de calcul est modélisé par la fonction *Step*, qui prend en paramètres les entrées de SHA et la valeur des objets locaux et de sortie, *LOCALS*, au cycle d'horloge t, et produit la valeur de *LOCALS* au cycle d'horloge t+1 (t est naturel).

$$Step\ (\text{Input, LOCALS}) \stackrel{def}{=} \langle nextval\_s\ (\text{Input, LOCALS})\ \rangle_{s \in LOCALS}$$

nextval_s est la fonction qui modélise le changement de s après un cycle d'horloge, $s \in$ LOCALS. Le corps de la fonction *Step* est la composition des *nextval_s*.

La fonction du système que nous obtenons à partir des équations récurrentes est :

```
System_sha_top(List_Inputs, LOCALS) def=
        if empty (List_Inputs) then LOCALS
          else let* New_LOCALS be
                        Step (First (List_Inputs), LOCALS)
                    in
               System_sha_top (Rest (List_Inputs), New_LOCALS)
        fi
```

La fonction *System_sha_top* modélise le circuit SHA-1. Le problème est que le circuit (et donc la fonction) réalise son calcul en interaction avec une RAM. Ainsi il faut ajouter la RAM dans le modèle et créer les fonctions nécessaires pour que le circuit et la mémoire interagissent de manière correcte. Dans le même temps, nous ne voulons pas réduire la généralité de la preuve, en ajoutant un composant hardware. Nous avons donc décidé d'utiliser une mémoire abstraite : une liste de couples (adresse, valeur). Dans le même souci de généralité, nous considérons que la taille de la liste est arbitraire. La seule contrainte est d'avoir assez d'espace pour garder le message. Nous considérons qu'à chaque place de la mémoire se trouve un mot de n bits, n étant 32 dans le cas du SHA-1. Puisque la RAM est potentiellement modifiée à chaque itération de *System_sha_top*, il faudra l'ajouter en paramètre.

Nous décrivons à nouveau le système afin de prendre en compte la RAM. La nouvelle fonction prend en paramètre une séquence d'entrées et le nouvel état *St* qui est la concaténation de l'ancien état LOCALS avec la RAM.

```
St def= (LOCALS | RAM)
Sha_Vhdl (List_Inputs, St) def=
    if empty (List_Inputs) then St
      else let* New_LOCALS be
                    Step (Read-RAM (LOCALS, RAM) | First (List_Inputs),
                          LOCALS)
                New_RAM be Write-RAM (New_LOCALS, RAM)
                in
           Sha_Vhdl (Rest (List_Inputs), New_LOCALS | New_RAM)
    fi
```

List_Inputs est la liste des valeurs numériques ou symboliques pour les entrées de SHA à chaque cycle d'horloge. La longueur de *List_Inputs* donne le nombre de cycles d'horloge pour lesquels la fonction s'exécute. Si la liste d'entrées est vide, l'exécution est finie, et l'état du système est *St*. Autrement, l'état après un cycle d'horloge est calculé et l'exécution continue avec le nouvel état. Encore un fois, ce modèle est exécutable, et nous avons simulé le modèle sur les tests fournis dans la documentation du standard.

La RAM est modélisée par une liste de couples ⟨*address*, mot de 32 bits⟩ où *address* est une valeur symbolique. Afin de garder la généralité de la preuve, la RAM est considérée

comme étant la concaténation de deux mémoires : une partie qui ne présente pas d'intérêt pour notre calcul, et une deuxième partie qui commence à partir de l'adresse *base_address*. Les adresses des deux parties sont disjointes.

4.3.3 Preuve de correction d'implémentation vs spécification

Maintenant, nous disposons de deux modèles de SHA-1 dans le démonstrateur ACL2 : la traduction du standard *Sha_Spec* qui n'a pas d'information temporelle, et la traduction automatique de la description VHDL, *Sha_Vhdl*, qui est au niveau du cycle d'horloge. Nous rappelons que *Sha_Vhdl* prend en entrée le message après rembourrage, contrairement au *Sha_Spec* qui calcule le message condensé à partir du message initial. Ainsi, nous allons comparer *Sha_Vhdl* avec *Digest* au lieu de *Sha_Spec*.

Pour montrer la correction de l'implémentation par rapport à la spécification, il faut montrer que pour tout message d'entrée, l'exécution de *Sha_Vhdl* pendant le nombre de cycles nécessaire pour finir le calcul, retourne le même message condensé que *Digest-Spec*.

Conformément à la méthodologie de vérification, nous définissons tout d'abord une abstraction comportementale du circuit : *Constrained*(*List_Inputs*). L'environnement attendu pour le SHA est présenté dans le Tableau 4.3, où X représente une valeur quelconque, *nb_block* est la représentation sur 6 bits du naturel *nb* : le nombre des blocs à traiter, *base_address* est un vecteur de bits de dimension 12. *nb_block* et *base_address* sont symboliques. Les deux premiers cycles initialisent le circuit et le calcul commence. A partir du troisième cycle, Reset est mis à 0, et les signaux *nb_block* et *base_address* sont stables. Ainsi la preuve est réalisée pour l'entrée :

$$\text{List\_Inputs} \stackrel{def}{=} (\langle \text{input\_cycle\_1, input\_cycle\_2} \rangle \mid \text{L\_input}).$$

Cycle	**1**	**2**	**3**	...
Input	*input_cycle_1*	*input_cycle_2*	L_Input	
Reset	1	0	0	...
Start	X	1	X	...
Reset_done	X	X	X	...
Nb_block	X	nb_block	nb_block	...
Base_address	X	X	base_address	...

TAB. 4.3 – L'entrée symbolique pour *Sha_Vhdl*

Dans un deuxième temps, il faut définir l'abstraction de données pour les entrées des deux modèles : le prédicat *Abstraction*. L'entrée pour la fonction *Digest* de la spécification de SHA-1 est la liste de *nb_block* blocs du message initial et la valeur initiale de hachage, qui est une constante de l'algorithme dans le cas de SHA-1. Le message à compresser se trouve dans la RAM à l'adresse *base_address*. Soit *Get-Message-from-RAM*(n, RAM, A) la fonction qui prend n * 16 mots de 32 bits stockés dans la RAM à l'adresse A, et retourne leur concaténation (car un bloc a 512 bits, et la mémoire stocke des mots de 32 bits). Ainsi, le prédicat d'abstraction *Abstraction* est défini de la manière suivante (*Parsing* est

la fonction d'analyse de la spécification) :

Abstraction(List_Inputs, RAM, Message) $\stackrel{def}{=}$ Message = *Parsing*(*Get-Message-from-RAM* (*nb_block*, RAM, *base_address*))

L'abstraction de données doit être aussi définie pour les sorties du circuit et le résultat de *Digest*. Dans notre cas, la concaténation des valeurs de signaux de sortie aout, bout, cout, dout, eout représente le condensé du message. Soit ABC_Vars(St) une fonction qui extrait de l'état les valeurs de cinq registres et retourne leur concaténation. Ainsi, le prédicat qui définit l'abstraction de sorties, *Result*, a la forme :

Result(St, digest)$\stackrel{def}{=}$ ABC_Vars(St) = *digest*

Le prédicat *Result* doit être vrai dans l'état final du circuit, qui est défini par :

Final(St) $\stackrel{def}{=}$ done=1

Pour énoncer le théorème de correction du circuit par rapport à l'implémentation, il reste à calculer le nombre de pas nécessaires au circuit pour finir le calcul.

Sha_Vhdl a besoin de 3 cycles d'horloge afin d'initialiser le système et d'initialiser les variables A, B, C, D, E avec les constantes de l'algorithme ; ensuite, 342 cycles d'horloge sont nécessaires afin de calculer le message condensé pour un bloc. Ce calcul est décomposé en plusieurs étapes : la lecture des 16 premiers mots et le calcul de 16 pas de l'algorithme nécessitent 16 cycles d'horloge ; ensuite, 320 cycles sont nécessaires pour calculer le reste de 64 pas de l'algorithme, 3 cycles pour combiner le résultat avec la valeur initiale de hachage, et 2 cycles pour mémoriser le résultat obtenu. Le dernier cycle retourne au calcul du bloc suivant ou à l'état *idle* s'il n'y a plus de bloc à traiter. Donc, pour calculer le message condensé pour un message de *nb_blocks* blocs, la description VHDL a besoin de $3 + (342 * nb\_blocks)$ cycles d'horloge.

Maintenant nous avons modélisé tous les éléments nécessaires à l'énoncé du théorème de correction :

Théorème de correction

Théorème 5 *(Correction) A partir d'un état quelconque, pour tout message de nb blocs stocké dans la RAM à l'adresse base_address, après l'exécution de Sha_Vhdl pendant* $3 + 342 * nb$ *cycles, le circuit a fini le calcul du message condensé, et la valeur calculée est égale au résultat de la fonction de spécification Digest appliquée au même message. nb est la conversion en entier de la valeur du signal nb_blocks.*

$\forall$ St = (LOCALS, RAM) $\wedge$
$\forall$ List_Inputs. *Constrained*(List_Inputs) $\wedge$
$\forall$ Message. *Abstraction*(List_Inputs, RAM, Message) $\wedge$

St_{new} = *Sha_Vhdl* (*First* (3+342*nb, List_Inputs), St)
Implies
Final(St_{new}) ∧ *Result* (St_{new}, *Digest* (Message, Initial_Hash_Val))

Pour prouver ce théorème, il faut prouver des théorèmes intermédiaires sur le comportement de *Sha_Vhdl* dans chaque étape de calcul. Les étapes de calcul sont celles définies dans l'automate VHDL du circuit (figure 4.4). Par contre, pour prouver les théorèmes intermédiaires, il faut définir les points de contrôle du circuit. Dans le cas de SHA-1, il n'est pas facile d'écrire une relation directe entre chaque étape de calcul et les fonctions correspondantes de la spécification. Ainsi, nous définissons des fonctions intermédiaires avec la même structure que les fonctions de la spécification, mais ayant comme entrées des objets de la description VHDL : *Digest-one-block-Impl* pour le calcul d'un bloc, et *Digest-Impl* pour le calcul complet. En effet, chaque fonction intermédiaire correspond à une abstraction comportementale de la description VHDL.

Les fonctions intermédiaires

La fonction *Digest-one-block-Impl* calcule le condensé du bloc courant, qui est stocké dans la mémoire RAM, à l'adresse *base_address + (nb_block - blocks_left) * 16).*

Digest-one-block-Impl (ABC_Vars, RAM, base_address, nb_block, blocks_left) $\stackrel{def}{=}$
Update (ABC_Vars,
Digest-step-Impl (80, '00000000', ABC_Vars, RAM, base_address,
nb_blocks, blocks_left))

La fonction *Digest-step-Impl* calcule i pas de l'algorithme à partir du pas *count*. A chaque pas, des opérations sont appliquées sur les registres a, b, c, d, e, groupées dans la variable ABC_Vars. Le bloc est aussi modifié par des opérations d'écriture dans la mémoire.

Digest-step-Impl (i, count, ABC_Vars, RAM, base_address, nb_block, blocks_left) $\stackrel{def}{=}$
if *Is_zero* (i) **then** ABC_Vars
else let* ABC_Vars **be** $Operations_{hash}$(count, ABC_Vars, RAM, base_address,
nb_block, blocks_left)
New-RAM **be** $Operations_{block}$(count, RAM, base_address, nb_block, blocks_left)
in *Digest-step-Impl* (i-1, count+1, ABC_Vars, New-RAM,
base_address, nb_block, blocks_left)
fi

Les opérations sur les valeurs de hachage sont les mêmes que dans la spécification. Pour le calcul de la variable TEMP, (*Temp-impl*), est utilisé le mot $W_{count\ mod\ 16}$. L'adresse du mot $W_{i+count\ mod\ 16}$ par rapport à *base_address* est calculée par la fonction *New-address*. Si $count < 16$, alors le mot $W_{count\ mod\ 16}$ est lu à partir de la RAM, sinon il est calculé en utilisant des mots correspondants lus à partir de la RAM (Comp-Word).

$Operations_{hash}$(count, ABC_Vars, RAM, base_address, nb_block, blocks_left) $\stackrel{def}{=}$

let* (a, b, c, d, e) **be** ABC_Vars
Address **be** *New-address* (nb_block, blocks_left, base_address, count, 0)
Comp-Word **be** *Word-Impl* (count, base_address, nb_block, blocks_left, RAM)
Word-from-RAM **be if** $16 \leq count$ **then** Comp-Word
else *Get*(Address, RAM) **fi**
New-a **be** *Temp-Impl* (count, a, b, c, d, e, Word-from-RAM)
New-c **be** Next-b(b)
in List(New-a, a, New-c, c, d)

La fonction *Word-Impl* calcule le nouveau $W_{count \bmod 16}$ et *Get* lit un mot à partir de la RAM.

Temp-Impl(count, a, b, c, d, e, data) $\stackrel{def}{=}$
F-impl (count, b, c, d) + e + data + *Rotl-Impl* (5, a) + K (count)

New-address (nb_block, blocks_left, base_address, count, i) $\stackrel{def}{=}$
base_address + i + *Segment* (4, 8, count) +
16 * (nb_block - blocks_left)

Word-Impl (count, base_address, nb_block, blocks_left, RAM) $\stackrel{def}{=}$
Rotl-Impl (1, *B-Xor*(
Get (*New-address* (nb_block, blocks_left, base_address, count, 0), RAM),
Get (*New-address* (nb_block, blocks_left, base_address, count, 2), RAM),
Get (*New-address* (nb_block, blocks_left, base_address, count, 8), RAM),
Get (*New-address* (nb_block, blocks_left, base_address, count, 13), RAM)))

F-impl et *Rotl-Impl* sont la fonction logique F et l'opération de rotation à gauche définies dans l'implémentation. Les opérateurs arithmétiques + et - sont surchargés pour prendre comme entrées des vecteurs de bit, ou un vecteur de bit et un entier positif.

Après le calcul des valeurs des registres a, b, c, d, e, la RAM est mise à jour. Si $count < 16$ alors la RAM n'est pas modifiée, sinon le mot $W_{count \bmod 16}$ est réécrit avec la nouvelle valeur Comp-Word.

$Operations_{block}$(count, RAM, base_address, nb_block, blocks_left) $\stackrel{def}{=}$
let* Address **be** *New-address* (nb_block, blocks_left, base_address, count, 0)
Comp-Word **be** *Word-Impl* (count, base_address, nb_block, blocks_left, RAM)
in if $16 \leq count$ **then** *Put* (Address, Comp-Word, RAM) **else** RAM **fi**

La preuve est compliquée par la présence de la RAM dans l'implémentation et l'ajout du contrôle pour accéder et écrire dans la mémoire des résultats partiels du calcul. En effet, un bloc de 512 bits est réécrit dans la mémoire quatre fois pendant le calcul, par les mots W_i, $(16 \leq i \leq 79)$. Ainsi, pour pouvoir raisonner sur la RAM, nous avons besoin de définir une fonction qui modélise comment la RAM est modifiée pendant l'exécution : *Modified-RAM*.

Tant que $count < 16$, il n'y a pas d'opération d'écriture sur la RAM. Si $16 \leq count$, à chaque pas *count* de l'algorithme, le mot $W_{count \bmod 16}$ est réécrit dans la RAM.

Modified-RAM (i, count, RAM, base_address, nb_blocks, blocks_left) $\stackrel{def}{=}$

if *Is_zero*(i) **then** RAM
Modified-RAM (i-1, count+1,
Operations$_{block}$(count, RAM, base_address, nb_block, blocks_left)
base_address, nb_blocks, blocks_left) **fi**

Digest-Impl abstrait le comportement général de la description VHDL. La fonction retourne les valeurs de hachage finales s'il n'existe plus de bloc à traiter. Sinon, le message condensé d'un bloc est calculé, les valeurs de hachage sont mises à jour, et le calcul continue avec le reste du message mémorisé dans la RAM.

Digest-Impl (Hash_Val, RAM, base_address, nb_blocks, blocks_left) $\stackrel{def}{=}$
if *Is_zero* (blocks_left) **then** Hash_Val
else *Digest-Impl* (*Digest-one-block-Impl* (80, '00000000', Hash_Val, RAM,
base_address, nb_blocks, blocks_left))
Modified-RAM(80, '00000000', RAM, base_address, nb_blocks, blocks_left)
base_address, nb_blocks, blocks_left-1))
fi

Digest-step-Impl et *Modified-RAM* ont une propriété de décomposition similaire, permettant la décomposition du raisonnement sur le comportement du circuit :

$\forall$ i, j naturels
Digest-step-Impl(i+j, count, ABC_Vars, RAM, params) =
Digest-step-Impl (i, count+j,
Digest-step-Impl (j, count, ABC_Vars, RAM, params)
Modified-RAM (j, count, RAM, params), params)

$\forall$ i, j naturels,
Modified-RAM (i+j, count, RAM, params) =
Modified-RAM (i, count+j, *Modified-RAM* (j, count, RAM, params), params)

Le comportement VHDL vs des fonctions intermédiaires

Pour prouver que l'implémentation est correcte par rapport à la spécification, nous allons d'abord prouver que les fonctions intermédiaires sont une abstraction comportementale et temporelle de l'implémentation. Et ensuite, nous allons prouver que la spécification est une abstraction de données et aussi comportementale des fonctions intermédiaires.

Pour la première étape (implémentation vs fonctions intermédiaires) nous allons suivre les pas de calcul mis en évidence par l'automate de contrôle du circuit VHDL (Figure 4.4). Ainsi, nous prouvons les théorèmes intermédiaires suivants :

- Les théorèmes 11 à 13 donnent le résultat de la phase d'initialisation.
- Le théorème 14 correspond aux 16 premiers pas de calcul. La RAM n'est pas modifiée.
- Le théorème 15 correspond aux 64 pas. Le bloc est réécrit dans la RAM.
- Les théorèmes 16 et 17 mettent à jour la valeur du message condensé pour un bloc et initialise le calcul pour le bloc suivant.
- Le théorème 18 combine les théorèmes 3 à 7 pour donner le résultat de calcul pour un bloc (correspondant aux 342 cycles d'horloge).

– Le théorème 19 combine les théorèmes 1,2 et 8 pour donner le résultat de calcul pour *nb_block* blocs (correspondant aux 3+342**nb_block* cycles).

Certains des théorèmes ci-dessus (théorèmes 4, 5, 8, 9) montrent que le circuit fait pendant un nombre déterminé de cycles d'horloge le même calcul que les fonctions intermédiaires. Ces théorèmes sont démontrés en utilisant la méthode de l'invariant présentée en début de chapitre.

Par la suite, les principaux théorèmes qui marquent les états de l'automate de contrôle du VHDL sont énumérés ; les détails de la preuve ACL2 se trouvent dans [125].

Théorème 6 *(D'un état arbitraire à idle)*
A partir d'un état quelconque, après un cycle, et ayant input_cycle_1 comme entrée, le système est dans l'état idle et la RAM n'est pas modifiée.

Théorème 7 *(De idle à init)*
A partir de l'état idle, après 2 cycles, le système est dans l'étape init, et les variables a, b ,c, d, e sont initialisées avec les constantes initiales de hachage, blocks_left est initialisé avec le nombre de blocs à traiter, i.e. nb_block, tous les autres objets du circuit sont initialisés et la RAM n'est pas modifiée.

Théorème 8 *(De init à SHA_init)*
A partir de l'étape init, après un cycle, le système est dans l'étape SHA_init (i.e. le calcul peut commencer), count est mis à 0 et la RAM n'est pas modifiée.

Les théorèmes ci-dessus sont obtenus par l'exécution symbolique de *Sha_Vhdl*.

Théorème 9 *(De SHA_init à compute_W)*
A partir de l'étape initiale de calcul, après 16 cycles, la RAM n'est toujours pas modifiée, et le système est dans l'étape compute_W, count est '00010000', et a, b, c, d, e ont la valeur des 16 premiers pas de calcul du Digest-step-Impl.

Dans la forme générale du théorème 14, le nombre fixe de cycles est généralisé avec $j \leq 16$, et *count* est un vecteur de 6 bits, avec la propriété : $bv - int(count) + j \leq 16$. Donc la forme généralisée montre qu'à partir d'un pas *count* quelconque de l'algorithme, après j cycles, le circuit calcule j pas d'algorithme de plus. Le théorème généralisé est prouvé par induction. Ensuite, j est instancié avec 16 et *count* avec '00000000'.

Théorème 10 *(De compute_W à result)*
A partir de l'étape du calcul pour le premier mot, après 320 cycles, la RAM est modifiée, le système est dans l'étape result, count est '01010000', et a, b, c, d, e ont le résultat des 64 derniers pas de calcul du Digest-step-Impl.

Les 5*64=320 cycles sont nécessaires pour calculer le message condensé d'un bloc : 5 cycles pour le calcul de TEMP (la variable intermédiaire) et 64 pour le nombre de fois

que l'algorithme doit être appliqué afin de calculer le condensé. Les deux étapes de calcul forment une boucle. Une itération de la boucle est réalisée en 5 cycles d'horloge.

Comme dans le cas précédent, il est d'abord prouvé une forme générale du théorème. Les paramètres généralisés sont les mêmes : *count* et le nombre j de cycles. Le nouveau théorème est prouvé par induction (un pas d'induction étant réalisé pour 5 cycles d'horloge). Ensuite, j est instancié avec 64 et *count* avec '00010000'.

Plusieurs lemmes sont nécessaires pour calculer le comportement de Sha_Vhdl pour 5 cycles, à partir de l'état $compute\_W$:

Lemma (De $compute\_W$ à $compute\_ABC$)

A partir d'une étape de calcul d'un mot, (étape = $compute\_W$), après 4 cycles, le mot correspondant $W_{count \bmod 16}$ est calculé par *Word-Impl*, et la nouvelle étape est $compute\_ABC$, *count* et la RAM ne changent pas.

Lemma (De $compute\_ABC$ à $compute\_W$ ou *result*)

A partir d'une étape de calcul de variables (étape = $compute\_ABC$, $count \leq 79$), après un cycle, si *count* est 79 alors l'algorithme est fini et la nouvelle étape est *result*, sinon, l'algorithme a été appliqué moins de 79 fois, et la nouvelle étape est $compute\_W$. Dans les deux cas, *count* est incrémenté, TEMP est calculé et $W_{count \bmod 16}$ est réécrit dans la RAM.

Théorème 11 *(De result à cnt_reset)*
A partir de l'étape result, après 3 cycles, le système est dans l'étape cnt_reset, le nombre de blocs à traiter est décrémenté et a, b, c, d, e sont additionnées aux h0, h1, h2, h3, h4, qui mémorisent les valeurs de hachage pendant le calcul.

Théorème 12 *(De cnt_reset à init ou idle)*
A partir de l'étape de reset, après 2 cycles, si le nombre de blocs à traiter est plus grand que 0, alors la nouvelle étape est init et count est mis à '00000000', sinon la nouvelle étape est idle, done est mis à 1 et les valeurs de a, b, c, d, e sont disponibles en sortie.

Le théorème suivant est obtenu en combinant les théorèmes 13 à 17.

Théorème 13 *(De init à init ou idle ⇔ Calcul de message condensé pour un bloc)*
A partir de l'étape initiale, après $1 + 16 + 320 + 3 + 2 = 342$ cycles d'horloge, si le nombre de blocs à traiter est plus grand que 0, alors le système est de nouveau dans l'étape initiale, sinon, le système est dans l'étape idle. Dans les deux cas la RAM est modifiée, et a, b, c, d, e ont la valeur du message condensé pour le bloc traité.

Théorème 14 *(Le circuit VHDL vs. La fonction intermédiaire)*
A partir d'un état quelconque, pour tout message de nb_blocks stocké dans la RAM à l'adresse base_address, après l'exécution de Sha_Vhdl pour $3 + 342 * nb\_blocks$ cycles, le système est dans son étape finale, (done = 1) et les valeurs de sortie ont la valeur du résultat de Digest-Impl appliquée au message.

$\forall$ St = (LOCALS, RAM) $\wedge$
$\forall$ List_Inputs. *Constrained*(List_Inputs) $\wedge$
St$_{new}$ = *Sha_Vhdl* (*First* (3+342*nb_blocks, List_Inputs), St)
Implies
Final(St$_{new}$) $\wedge$
Result (St$_{new}$, *Digest-Impl* (Initial_Hash_Val, RAM, base_address, nb_blocks, nb_blocks)

Tout d'abord, nous prouvons une forme généralisée du théorème qui démontre qu'à partir de l'étape d'initialisation, *init*, s'il reste j blocs à traiter, après $342 * j$ cycles, le circuit finit le calcul. Le théorème généralisé est prouvé par induction, un pas d'induction étant réalisé en 342 cycles d'horloge.

Ensuite, en utilisant le théorème 11 et le théorème 12, nous prouvons le théorème 19.

Des fonctions intermédiaires vs la spécification

Jusqu'à présent, nous avons prouvé que *Digest-Impl* est une abstraction correcte du comportement de *Sha_Vhdl*. Le pas suivant est de prouver que la spécification *Digest* est équivalente à la fonction intermédiaire pour le même message, tenant compte de l'éventuelle abstraction de données.

Théorème 15 *(Fonction intermédiaire vs. Spécification)*
Digest-Impl (Initial_Hash_Val, RAM, base_address, nb_blocks, nb_blocks) =
Digest (Parsing (Get-Message-from-RAM (nb_blocks, base_address, RAM), 512), Initial_Hash_Val)

Get-Message-from-RAM prend en entrée *nb_blocks* de 16 mots de 32 bits stockés dans la RAM à *Address* et retourne leur concaténation.

Afin de prouver ce théorème, il faut d'abord prouver une forme généralisée : l'implémentation et la spécification calculent le même message condensé pour des valeurs initiales de hachage arbitraires et pour un nombre $k = nb\_block - blocks\_left$ de blocs de message.

La preuve utilise le schéma d'induction généré par *Digest-Impl* et plusieurs lemmes. Nous mentionnons les plus importantes :

Lemme

Après le calcul du message condensé pour k blocs, la partie de la RAM qui stocke le reste du message (à partir de l'adresse $base\_address + 16 * k$) n'est pas modifiée.

Lemme

La spécification et l'implémentation calcule le même message condensé pour un bloc :
Digest-step-Impl
(80, '00000000', ABC_Vars, RAM, base_address, nb_blocks, blocks_left) =

Digest-step (0, ABC_Vars, (*Get-Block-from-RAM* (Address, RAM)))

Pour le bloc courant, Address = $base\_address + 16 * (nb\_block - blocks\_left)$.

La preuve utilise aussi les propriétés suivantes :

- la spécification et l'implémentation calculent le même mot : $W_{(count\ mod\ 16)}$.

- l'opération de rotation à gauche et la fonction logique F de l'implémentation sont égales à leurs correspondantes dans la spécification.

- le remplacement du mot $W_{(count\ mod\ 16)}$ avec le nouveau mot calculé a le même effet sur le message initial tant dans la spécification que dans l'implémentation.

Les théorèmes énoncés utilisent un grand nombre de propriétés sur les vecteurs de bits, sur les opérations sur les vecteurs de bits (logiques, arithmétiques, de concaténation, de conversion, shift, etc.). Nous avons aussi dû prouver des propriétés sur la RAM, et *List_inputs*. La bibliothèque sur les vecteurs de bits a 64 fonctions et 504 théorèmes. La preuve de SHA-1, les deux modèles inclus, a nécessité 162 fonctions et 653 théorèmes.

4.3.4 Bilan

Durant la preuve du SHA-1, nous avons trouvé quelques erreurs dans le circuit :

- une erreur d'optimisation : le test sur une variable du circuit n'était pas correct. Ceci entraînait l'exécution d'un nombre de cycles excédentaire, jusqu'au débordement de la variable et sa réinitialisation.

- des opérations illégales dans la mémoire : la mémoire était lue aussi dans des étapes de calcul qui n'utilisaient pas la donnée lue. Ceci ne changeait pas le résultat du calcul, mais l'opération étant coûteuse en temps, elle n'était pas désirable. Une autre erreur a été l'écriture de la mémoire dans des zones non concernées par le calcul courant. Plus précisément, des nouveaux mots étaient écrits dans une zone de mémoire qui correspondait aux blocs déjà traités par le circuit. Puisque le composant était censé être utilisé sur une puce où la mémoire qu'il accédait était partagée avec d'autres composants, la mise en évidence de cette erreur a été importante.

Conclusion et perspectives

Les systèmes électroniques sont devenus de plus en plus complexes et sont présents dans la plupart des produits que nous utilisons tous les jours. Pour certains de ces systèmes, ils mettent en jeu des vies humaines (peace maker, appareils d'opérations chirurgicales à distance, gestion du trafic aérien, système de refroidissement dans les centrales nucléaires, etc...). Le mauvais fonctionnement d'un circuit électronique peut donc avoir des conséquences financières et humaines très graves.

Assurer la correction des circuits dans des applications sécuritaires demande un processus de conception très rigoureux. *Idéalement*, une spécification fonctionnelle formelle est d'abord définie. Ensuite elle est validée, et des pas successifs de raffinement prouvés corrects sont appliqués, jusqu'au niveau de transfert de registres. A partir de ce niveau, les outils de synthèse produisent automatiquement le circuit physique. *En réalité*, la pression du marché est tellement grande que les logiciels d'aide à la conception doivent être faciles à utiliser et automatiques pour être effectivement adoptés. Les travaux de recherche présentés dans ce manuscrit cherchent à "pousser" les concepteurs vers cet idéal.

La simulation numérique va probablement rester la première méthode de validation de la conception, à tous les niveaux d'abstraction. En effet, un grand nombre d'erreurs est mis en évidence par simulation numérique ; mais pour garantir la correction du fonctionnement, les méthodes formelles doivent être utilisées soit pour valider les spécifications initiales soit pour vérifier un pas de raffinement.

Ce manuscrit de thèse présente notre contribution à la vérification formelle des systèmes numériques.

Parmi les méthodes de vérification présentées dans le premier chapitre, nous avons choisi la démonstration de théorèmes pour sa capacité de raisonnement sur des objets non bornés. Nous n'avons pas choisi les techniques de vérification formelle booléennes basées sur le parcours d'espace d'état, sur la procédure SAT, etc..., qui sont implémentées aujourd'hui dans des outils d'aide à la conception, car elles ne sont pas applicables dans une phase initiale du projet, où la spécification est exprimée en termes d'opérations arithmétiques et d'algorithmes de plus haut niveau d'abstraction. En effet, à ce niveau, les types de données ne sont pas bornés et le modèle n'a pas une taille fixe.

Parmi les démonstrateurs de théorèmes existants, nous avons choisi ACL2, pour sa capacité à réutiliser des bibliothèques de fonctions et théorèmes pré-vérifiés, son degré

d'automatisation et l'efficacité de son moteur de preuve. Le modèle ACL2 étant écrit en Lisp est exécutable mais aussi prouvable dans la logique du démonstrateur.

Notre méthodologie de vérification formelle des systèmes numériques se décompose en deux étapes. Dans la première, nous validons la spécification par des tests numériques et la vérification des propriétés associées à cette spécification. A la fin de cette étape, nous sommes assurés d'avoir une spécification correcte.

La deuxième étape consiste à prouver que l'implémentation correspond à la spécification. Il n'existe pas une relation d'équivalence directe, puisqu'un raffinement de temps, de données, de structure a enrichi le modèle. Pour réaliser cette vérification, il faut extraire un modèle fonctionnel à partir de l'implémentation. Cette seconde étape de notre méthode de vérification a nécessité que nous prenions en compte les descriptions matérielles des systèmes numériques.

Comme langage de description matérielle, nous avons considéré VHDL. Nous avons pris en compte un sous-ensemble de VHDL contenant les boucles, les sous-programmes et les types composés. Par la suite, nous avons présenté les fondements sémantiques de la méthode d'extraction d'un modèle fonctionnel à partir d'une description matérielle en VHDL.

Le modèle fonctionnel que nous proposons (chapitre 2) est basé sur des équations récurrentes. Ce formalisme, étendu aux équations récurrentes affines, a déjà été appliqué à la synthèse des architectures systoliques [105]. Initialement, les algorithmes sont spécifiés par des équations mutuellement récurrentes. La spécification est raffinée jusqu'à une forme qui permet la génération d'un circuit régulier. Le résultat de la synthèse est correct par construction par rapport à la spécification, donc aucune vérification supplémentaire n'est nécessaire.

Notre démarche est ascendante. A partir d'une description VHDL, pas nécessairement régulière, nous obtenons un système d'équations récurrentes. Les équations que nous obtenons sont beaucoup plus simples que celles utilisées dans l'approche précédente, étant récurrentes seulement par rapport au temps et non par rapport à l'espace. Le système peut être ensuite simulé symboliquement afin d'obtenir des comportements différents de circuit. Nous avons également étudié la faisabilité de la vérification du système par la démonstration de théorèmes.

Nous avons proposé une méthode de traduction automatique de VHDL vers un démonstrateur de théorèmes avec une implémentation pour ACL2 (chapitre 3). Finalement, nous avons illustré l'approche sur des composants cryptographiques (chapitre 4).

Perspectives

Notre approche de vérification s'est montrée efficace pour les composants matériels au niveau RTL. Une perspective du travail est donc de l'étendre au niveau système.

Le projet PUSSEE [90] a défini une méthodologie descendante (*top-down*) basée sur la preuve formelle pour développer des systèmes électroniques par raffinement à partir d'un modèle très abstrait jusqu'au niveau transfert de registre, suivie d'une traduction vers

des langages de description matérielle. Event-B [2, 1] est utilisé comme un environnement formel et BHDL est le niveau d'implémentation défini en B pour les circuits électroniques.

Un inconvénient de cette dernière approche est qu'elle ne permet pas la réutilisation des composants. Pour assurer la correction du résultat, les composants doivent être développés à l'intérieur de la méthode B et traduits ensuite dans un langage de description matérielle. La réutilisation des composants existants demande le parcours à l'envers. Il est possible de traduire la description matérielle du composant en B et d'effectuer des preuves sur le modèle B [3], mais il est difficile à l'intérieur de la méthode de réaliser la preuve, puisque généralement il y a beaucoup de différences entre l'abstraction et le modèle B de l'implémentation, et donc, plusieurs raffinements sont demandés afin de rendre les preuves possibles.

Dans [141] nous avons présenté une alternative à cette approche : le circuit est spécifié en B au niveau d'abstraction où l'interface du modèle correspond à l'interface du circuit. Ensuite le modèle B est traduit en ACL2. Dans le même temps la description VHDL est traduite aussi en ACL2 en utilisant notre méthode. ACL2 est utilisé pour prouver l'équivalence entre les deux modèles.

Le modèle formel des systèmes électroniques que nous avons développé est adapté à tous les démonstrateurs de théorèmes qui disposent comme technique de preuve de la preuve par récurrence. Dans cette thèse, nous avons choisi d'utiliser ACL2, mais ce travail peut être porté dans d'autres démonstrateurs. C'est une perspective de travail que nous prenons en compte afin de diffuser les résultats de nos travaux.

Une amorce de ce travail a débuté dans notre équipe avec le démonstrateur PVS : les équations récurrentes étant traduites comme des fonctions dépendantes du temps (comme cela a été présenté dans le chapitre 3). PVS est basé sur une logique d'ordre supérieur permettant une quantification sur les fonctions. Dans ce cas l'encapsulation d'ACL2 est évitée.

Le même approche permettrait de prendre en compte d'autres langages de description matérielle que VHDL. Pour cela il faudrait réécrire la première phase de compilation et adapter l'algorithme de simulation. Les règles de transformation des instructions et les règles de réécriture construites dans le simulateur symbolique restent valables. En effet les règles de transformation correspondent à la sémantique classique des concepts communs à tous les langages de description matérielle. Dans tous ces langages on trouve les instructions séquentielles (de contrôle, de boucle, d'affectation), la distinction entre variables et signaux, l'instruction de synchronisation, le parallélisme, les types scalaires et composés, l'interconnexion de composants.

Une dernière perspective de notre travail est d'appliquer la méthodologie de preuve sur d'autres types de circuits. Nous avons pris comme étude de cas des circuits cryptographiques mais notre approche n'est pas caractéristique à cette famille. En effet, la même approche a été appliquée dans notre équipe pour la preuve des moniteurs VHDL [94]. Les familles de circuits auxquelles notre méthodologie de vérification s'applique d'une manière efficace sont les circuits parametrées, le circuits de traitement de signal, et generalement les circuits ayant une partie donnée très importante.

Chapitre 5

ANNEXE : La syntaxe du sous-ensemble VHDL

La syntaxe du sous-ensemble VHDL

```
abstract_literal : := decimal_literal | based_literal
access_type_definition : := access subtype_indication
actual_designator : :=
      expression
      | signal_name
      | variable_name
      | file_name
      | open
actual_parameter_part : := parameter_association_list
actual_part : :=
      actual_designator
      | function_name ( actual_designator )
      | type_mark ( actual_designator )
      adding_operator : := + | - | &
architecture_body : :=
       architectureidentifier  of entity_name  is
      architecture_declarative_part
       begin
      architecture_statement_part
       end [ architecture] [architecture_simple_name] ;
architecture_declarative_part : :=  block_declarative_item
architecture_statement_part : :=  concurrent_statement
array_type_definition : :=
      unconstrained_array_definition | constrained_array_definition
association_element : :=
      [formal_part =>] actual_part
```

```
      association_list : :=
      association_element  , association_element
base : := integer
base_specifier : := B | O | X
base_unit_declaration : := identifier ;
based_integer : := extended_digit  [underline] extended_digit
based_literal : := base # based_integer [. based_integer] # [exponent]
basic_character : := basic_graphic_character | format_effector
basic_graphic_character : :=
     upper_case_letter | digit | special_character| space_character
basic_identifier : := letter  [underline] letter_or_digit
binding_indication : :=
     [use entity_aspect]
     [generic_map_aspect]
     [port_map_aspect]
bit_string_literal : := base_specifier " [bit_value] "
bit_value : := extended_digit  [underline] extended_digit
block_configuration : :=
      for block_specification
      use_clause
      configuration_item
      end  for ;
block_declarative_item : :=
     subprogram_declaration
     | subprogram_body
     | type_declaration
     | subtype_declaration
     | constant_declaration
     | signal_declaration
     | component_declaration
     | configuration_specification
     | use_clause
block_declarative_part : :=  block_declarative_item
block_header : :=
      [generic_clause
      [generic_map_aspect ;]]
      [port_clause
      [port_map_aspect ;]]
block_specification : :=
      architecture_name
      | block_statement_label
      | generate_statement_label [( index_specification )]
```

```
block_statement : :=
      block_label :
       block [( guard_expression )] [ is]
      block_header
      block_declarative_part
       begin
      block_statement_part
       end  block [block_label] ;
block_statement_part : :=  concurrent_statement
case_statement : :=
      [case_label :]
       case expression  is
      case_statement_alternative
       case_statement_alternative
       end  case [case_label] ;
case_statement_alternative : :=
       when choices =>
      sequence_of_statements
      character_literal : := ' graphic_character '
choice : :=
      simple_expression
      | discrete_range
      | element_simple_name
      |  others
choices : := choice  | choice
component_configuration : :=
       for component_specification
      [binding_indication ;]
      [block_configuration]
       end  for ;
component_declaration : :=
       component identifier [ is]
      [local_generic_clause]
      [local_port_clause]
       end  component [component_simple_name] ;
component_instantiation_statement : :=
      instantiation_label :
      instantiated_unit
      [generic_map_aspect]
      [port_map_aspect] ;
component_specification : := instantiation_list : component_name
composite_type_definition : :=
```

```
      array_type_definition
      | record_type_definition
concurrent_procedure_call_statement : := [label :] procedure_call ;
concurrent_signal_assignment_statement : :=
      [label :] conditional_signal_assignment
      | [label :] selected_signal_assignment
concurrent_statement : :=
      block_statement
      | process_statement
      | concurrent_procedure_call_statement
      | concurrent_signal_assignment_statement
      | component_instantiation_statement
      | generate_statement
condition : := boolean_expression
condition_clause : := until condition
conditional_signal_assignment : := target <= conditional_waveforms ;
conditional_waveforms : :=
       waveform when condition else
      waveform [ when condition]
configuration_declaration : :=
       configuration identifier of entity_name is
      configuration_declarative_part
      block_configuration
       end [ configuration] [configuration_simple_name] ;
configuration_declarative_item : :=
      use_clause
      | attribute_specification
      | group_declaration
configuration_declarative_part : := configuration_declarative_item
configuration_item : :=
      block_configuration
      | component_configuration
configuration_specification : :=
       for component_specification binding_indication ;
constant_declaration : :=
      constant identifier_list : subtype_indication [ := expression] ;
constrained_array_definition : :=
       array index_constraint of element_subtype_indication
constraint : := range_constraint | index_constraint
context_clause : := context_item
context_item : := library_clause | use_clause
decimal_literal : := integer [. integer] [exponent]
```

```
declaration : :=
      type_declaration
      | subtype_declaration
      | object_declaration
      | interface_declaration
      | component_declaration
      | entity_declaration
      | configuration_declaration
      | subprogram_declaration
      | package_declaration
design_file : := design_unit  design_unit
design_unit : := context_clause library_unit
designator : := identifier | operator_symbol
direction : :=  to |  downto
discrete_range : := discrete_subtype_indication | range
element_association : := [choices =>] expression
element_declaration : := identifier_list : element_subtype_definition ;
element_subtype_definition : := subtype_indication
entity_aspect : :=
      entity entity_name [( architecture_identifier)]
      | configuration configuration_name
entity_class : :=
      entity | architecture | configuration
      | procedure | function | package
      | type | subtype | constant
      | signal | variable | component
      | label | literal
entity_class_entry : := entity_class [<>]
entity_class_entry_list : := entity_class_entry  , entity_class_entry
entity_declaration : :=
       entity identifier  is
      entity_header
      entity_declarative_part
      [ begin
      entity_statement_part]
       end [ entity] [entity_simple_name] ;
entity_declarative_item : :=
      subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
```

```
      | signal_declaration
      | use_clause
entity_declarative_part : :=  entity_declarative_item
entity_designator : := entity_tag [signature]
entity_header : :=
      [formal_generic_clause]
      [formal_port_clause]
entity_name_list : :=
      entity_designator  , entity_designator
      |  others
      |  all
entity_specification : := entity_name_list : entity_class
entity_statement : :=
      concurrent_assertion_statement
      | passive_concurrent_procedure_call_statement
      | passive_process_statement
entity_statement_part : :=  entity_statement
entity_tag : := simple_name | character_literal | operator_symbol
enumeration_literal : := identifier | character_literal
enumeration_type_definition : :=
      ( enumeration_literal  , enumeration_literal  )
exponent : := E [+] integer | E - integer
expression : :=
      relation  and relation
      | relation  or relation
      | relation  xor relation
      | relation [nand relation]
      | relation [nor relation]
      | relation  xnor relation
extended_digit : := digit | letter
extended_identifier : :=   graphic_character  graphic_character
factor : :=
      primary [** primary]
      | abs primary
      | not primary
floating_type_definition : := range_constraint
formal_designator : :=
      generic_name
      | port_name
      | parameter_name
formal_parameter_list : := parameter_interface_list
formal_part : :=
```

```
      formal_designator
      | function_name ( formal_designator )
      | type_mark ( formal_designator )
function_call : :=  function_name [( actual_parameter_part )]
generate_statement : :=
      generate_label :
      generation_scheme  generate
      [ block_declarative_item
       begin]
        concurrent_statement
       end generate [generate_label] ;
generation_scheme : :=
       for generate_parameter_specification
      |  if condition
generic_clause : :=  generic ( generic_list ) ;
generic_list : := generic_interface_list
generic_map_aspect : :=  generic map ( generic_association_list )
graphic_character : :=
      basic_graphic_character | lower_case_letter | other_special_character
guarded_signal_specification : := guarded_signal_list : type_mark
identifier : := basic_identifier | extended_identifier
identifier_list : := identifier  , identifier
if_statement : :=
      [if_label :]
       if condition  then
      sequence_of_statements
        elsif condition  then
      sequence_of_statements
      [ else
      sequence_of_statements]
       end  if [if_label] ;
index_constraint : := ( discrete_range  , discrete_range  )
index_specification : :=
      discrete_range
      | static_expression
      index_subtype_definition : := type_mark range <>
indexed_name : := prefix ( expression  , expression  )
instantiated_unit : :=
      [component] component_name
      | entity entity_name [( architecture_identifier )]
      | configuration configuration_name
instantiation_list : :=
```

```
      instantiation_label  , instantiation_label
      | others
      | all
integer : := digit  [underline] digit
integer_type_definition : := range_constraint
interface_constant_declaration : :=
      [ constant] identifier_list :
        [ in] subtype_indication [ := static_expression]
interface_declaration : :=
      interface_constant_declaration
      | interface_signal_declaration
      | interface_variable_declaration
interface_element : := interface_declaration
interface_list : := interface_element  ; interface_element
interface_signal_declaration : :=
      [ signal] identifier_list :
        [mode] subtype_indication [bus] [ := static_expression]
interface_variable_declaration : :=
      [ variable] identifier_list :
        [mode] subtype_indication [ := static_expression]
iteration_scheme : :=
       while condition
      |  for loop_parameter_specification
label : := identifier
letter : := upper_case_letter | lower_case_letter
letter_or_digit : := letter | digit
library_clause : := library logical_name_list ;
library_unit : := primary_unit | secondary_unit
literal : :=
      numeric_literal
      | enumeration_literal
      | string_literal
      | bit_string_literal
      | null
logical_name : := identifier
logical_name_list : := logical_name  , logical_name
logical_operator : := and | or | nand | nor | xor | xnor
loop_statement : :=
      [loop_label :]
      [iteration_scheme]  loop
      sequence_of_statements
       end loop [loop_label] ;
```

```
miscellaneous_operator : := ** | abs | not
mode : := in | out
multiplying_operator : := * | / | mod | rem
name : :=
      simple_name
      | operator_symbol
      | selected_name
      | indexed_name
      | slice_name
null_statement : := [label :]  null ;
numeric_literal : := abstract_literal
object_declaration : :=
      constant_declaration
      | signal_declaration
      | variable_declaration
operator_symbol : := string_literal
package_body : :=
       package body package_simple_name  is
      package_body_declarative_part
       end [ package body] [package_simple_name] ;
package_body_declarative_item : :=
      subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | shared_variable_declaration
      | file_declaration
      | alias_declaration
      | use_clause
      | group_template_declaration
      | group_declaration
package_body_declarative_part : :=  package_body_declarative_item
package_declaration : :=
       package identifier  is
      package_declarative_part
       end [ package] [package_simple_name] ;
package_declarative_item : :=
      subprogram_declaration
      | type_declaration
      | subtype_declaration
      | constant_declaration
```

```
      | signal_declaration
      | use_clause
package_declarative_part : :=  package_declarative_item
parameter_specification : := identifier  in discrete_range
port_clause : :=  port ( port_list ) ;
port_list : := port_interface_list
port_map_aspect : :=  port map ( port_association_list )
prefix : := name | function_call
primary : :=
      name
      | literal
      | aggregate
      | function_call
      | qualified_expression
      | type_conversion
      | allocator
      | ( expression )
primary_unit : :=
      entity_declaration
      | configuration_declaration
      | package_declaration
procedure_call : := procedure_name [( actual_parameter_part )]
procedure_call_statement : := [label :] procedure_call ;
process_declarative_item : :=
      subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | variable_declaration
      | use_clause
process_declarative_part : :=  process_declarative_item
process_statement : :=
      [process_label :]  process [( sensitivity_list )] [ is]
      process_declarative_part
       begin
      process_statement_part
       end process [process_label] ;
process_statement_part : :=  sequential_statement
range : :=
      range_attribute_name
      | simple_expression direction simple_expression
```

```
range_constraint : := range
record_type_definition : :=
       record
      element_declaration
       element_declaration
       end record [record_type_simple_name]
relation : := shift_expression [relational_operator shift_expression]
relational_operator : := = | /= | < | <= | > | >=
return_statement : := [label :] return [expression] ;
scalar_type_definition : :=
      enumeration_type_definition | integer_type_definition
      | floating_type_definition
secondary_unit : := architecture_body | package_body
selected_name : := prefix . suffix
selected_signal_assignment : :=
       with expression  select
      target <= selected_waveforms ;
selected_waveforms : :=
       waveform  when choices ,
      waveform  when choices
sensitivity_clause : :=  on sensitivity_list
sensitivity_list : := signal_name  , signal_name
sequence_of_statements : :=
       sequential_statement
sequential_statement : :=
      wait_statement
      | signal_assignment_statement
      | variable_assignment_statement
      | procedure_call_statement
      | if_statement
      | case_statement
      | loop_statement
      | return_statement
      | null_statement
shift_expression : := simple_expression [shift_operator simple_expression]
shift_operator : := sll | srl | sla | sra | rol | ror
sign : := + | -
signal_assignment_statement : := [label :] target <= waveform ;
signal_declaration : :=
      signal identifier_list : subtype_indication [ := expression] ;
signal_list : :=
      signal_name  , signal_name
```

```
      | others
      | all
signature : := [[type_mark  , type_mark ] [ return type_mark]]
simple_expression : := [sign] term  adding_operator term
simple_name : := identifier
slice_name : := prefix ( discrete_range )
string_literal : := " graphic_character  " "
subprogram_body : :=
      subprogram_specification  is
      subprogram_declarative_part
       begin
      subprogram_statement_part
       end [subprogram_kind] [designator] ;
subprogram_declaration : := subprogram_specification ;
subprogram_declarative_item : :=
      subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | variable_declaration
      | use_clause
subprogram_declarative_part : :=  subprogram_declarative_item
subprogram_kind : :=  procedure |  function
subprogram_specification : :=
       procedure designator [( formal_parameter_list )]
      | [pure]  function designator [( formal_parameter_list )]
       return type_mark
subprogram_statement_part : :=  sequential_statement
subtype_declaration : :=  subtype identifier  is subtype_indication ;
subtype_indication : := [resolution_function_name] type_mark [constraint]
suffix : :=
      simple_name
      | character_literal
      | operator_symbol
      | all
target : := name | aggregate
term : := factor multiplying_operator factor
type_conversion : := type_mark ( expression )
type_declaration : :=  type identifier  is type_definition ;
type_definition : :=
      scalar_type_definition
```

```
      | composite_type_definition
type_mark : :=
      type_name
      | subtype_name
unconstrained_array_definition : :=
       array ( index_subtype_definition  , index_subtype_definition  )
       of element_subtype_indication
use_clause : :=  use selected_name  , selected_name  ;
variable_assignment_statement : := [label :] target := expression ;
variable_declaration : :=
       variable identifier_list : subtype_indication [ := expression] ;
wait_statement : :=
      [label :]  wait [sensitivity_clause] [condition_clause] ;
waveform : := value_expression
```

Chapitre 6

ANNEXE : Les transformations des instructions VHDL

Trans(**signal** *name* : type :=val) $\triangleq$ *name* := **Trans**(val)
Trans(**variable** *name* : type :=val) $\triangleq$ *name* := **Trans**(val)
Trans(**constant** *name* : type :=val) $\triangleq$ *name* := **Trans**(val)

Trans (type **function** *name* (params) Dec Seq **return** *expression*) $\triangleq$
name (params) = $expression[x/extract(x, \textbf{Trans}(\text{Decl}\,;\,\text{Seq}))]$
$\forall x \in \text{R}_e xpression \cap \text{W}_{\text{Seq}}$

Trans (**procedure** *name* (params) Decl ; Seq) $\triangleq$
name (params) = $\|_{x \in params \cap \text{W}_{\text{Seq}}} \{x := extract(x, \textbf{Trans}(\text{Decl}\,;\,\text{Seq}))\}$

TAB. 6.1 – Transformation des déclarations

$$\textbf{Trans}(n) \triangleq n$$
$$\textbf{Trans}(id) \triangleq nth(pos(id), \mathrm{W}_{id})$$
$$\textbf{Trans}(function\_call) \triangleq function\_call$$
$$\textbf{Trans}((e)) \triangleq (\textbf{Trans}(e))$$
$$\textbf{Trans}(\text{op } e) \triangleq \text{op } \textbf{Trans}(e)$$
$$\textbf{Trans}(e \text{ op } f) \triangleq \textbf{Trans}(e) \text{ op } \textbf{Trans}(f)$$
$$\textbf{Trans}(\textbf{if } b \textbf{ then } expression_1 \textbf{ endif}) \triangleq IF(\textbf{Trans}(b), \textbf{Trans}(expression_1), undef)$$
$$\textbf{Trans } (\textbf{if } b \textbf{ then } expression_1 \textbf{ else } expression_2 \textbf{ endif}) \triangleq IF(\textbf{Trans}(b), \textbf{Trans}(expression_1), \textbf{Trans}(expression_2))$$

TAB. 6.2 – Transformation des expressions

Trans ($id <= expression$) $\triangleq$
$\quad \mathrm{W}_{id} := replace\ ((pos(id),$ **Trans** $(expression)[undef/\widetilde{\mathrm{W}}_{id}]), \widetilde{\mathrm{W}}_{id})$
Trans ($id := expression$) $\triangleq$
$\quad \mathrm{W}_{id} := replace((pos(id),$ **Trans**$(expression)[undef/\mathrm{W}_{id}]),\ \mathrm{W}_{id})$

Trans (**if** *condition* **then** Seq **endif**) $\triangleq$
$\quad \|_{x \in \mathrm{W_{Seq}}}$ { x := IF (**Trans**$(condition)$, $extract(x,$ **Trans**$(Seq))$,
$\qquad\qquad\qquad\qquad extract(x, \emptyset))$ }
Trans (**if** *condition* **then** $\mathrm{Seq_1}$**else** $\mathrm{Seq_2}$ **endif** $\triangleq$
$\quad \|_{x \in \mathrm{W_{Seq_1}} \cup \mathrm{W_{Seq_2}}}$ { x := IF(**Trans**$(condition)$, $extract(x,$ **Trans**$(\mathrm{Seq_2}))$,
$\qquad\qquad\qquad\qquad\qquad extract(x,$ **Trans**$(\mathrm{Seq_1})))$ }

Trans (**while** *cond* **do** Seq **end do**) $\triangleq$
$\quad \|_{x \in \mathrm{W_{Seq}} \cap \mathcal{V}} \{x := IF(cond, while\_x(x, \mathrm{R}_{E_x} \cup \mathrm{R}_{cond}), x)\}$
$\quad \|_{x \in \mathrm{W_{Seq}} \cap \mathcal{S}} \{x := IF(cond, while\_x(x, \mathrm{R}_{E_x} \cup \mathrm{R}_{cond}), \widetilde{x})\}$

où $while\_x(x, x_1, \ldots, x_k) = IF(cond, while\_x(E_x[\widetilde{x}/x], F_{x_1}, \ldots, F_{x_k}), x)$
et $E_x = extract(x,$ **Trans**$(\mathrm{Seq}))$ et
$\mathrm{R}_{E_x} \cup \mathrm{R}_{cond} = \{x_1, \ldots, x_k\}$ et
F_{x_i} est E_{x_i} si $x_i \in \mathcal{V} \cap \mathrm{W_{Seq}} \cap (\mathrm{R}_{E_x} \cup \mathrm{R}_{cond})$, et x_i autrement.

Trans (**for** i **from** a_1 **to** a_2 **do** Seq **end for**) $\triangleq$
$\quad \|_{x \in \mathrm{W_{Seq}} \cap \mathcal{V}} \{x := IF(a_1 \leq a_2, for\_x(x, a_1, \mathrm{R}_{E_x} \cup \mathrm{R}_{a_2}), x)\}$
$\quad \|_{x \in \mathrm{W_{Seq}} \cap \mathcal{S}} \{x := IF(a_1 \leq a_2, for\_x(x, a_1, \mathrm{R}_{E_x} \cup \mathrm{R}_{a_2}), \widetilde{x})\}$

où $for\_x(x, i, x_1, \ldots, x_k) = IF(i \leq a_2, for\_x(E_x[\widetilde{x}/x], i+1, F_{x_1}, \ldots, F_{x_k}), x)$.

Trans ($\mathrm{Seq_1}$;$\mathrm{Seq_2}$) $\triangleq$
$\quad \|_{x \in \mathrm{W_{Seq_1}} - \mathrm{W_{Seq_2}}} \{x := extract(x, \mathbf{Trans}_{\mathrm{Seq_1}}\}$
$\quad \|substitute($**Trans**$(\mathrm{Seq_1}),$ **Trans**$(\mathrm{Seq_2}))$

TAB. 6.3 – Transformation des instructions séquentielles

Trans (id <= $expression$) $\triangleq$
$\quad$ W_{id} := IF ($Event$ ($Sensitivity$ ($expression$)),
$\quad\quad$ $replace((pos(id),$**Trans**$(expression)[undef/\mathrm{W}_{id}]), \widetilde{\mathrm{W}}_{id})$,
$\quad\quad$ W_{id})

Trans ($name$ (params) Ent ; Arch) $\triangleq$
$\quad$ $name\_Ent\_Arch$ (params $\cup \mathcal{L}ocals$) = $\|_{x \in \mathcal{L}ocals}\{x := extract(x,$ **Trans**(Conc))$\}$

Trans ($label$: **process** ($sensitivity\_list$) Decl **begin** Seq **end process**) $\triangleq$
$\quad$ **Trans** (**if** $Event$(sensitivity_list) **then** Seq **endif**)

Trans ($label$: **process** Decl **begin** Seq **end process**) $\triangleq$
$\quad$ **Trans** ($label$: **process** ($sensitivity\_list$)
$\quad\quad\quad\quad\quad\quad$ Decl ; **variable** pw : integer := 1 ;
$\quad\quad\quad\quad\quad\quad$ **begin** $Modify$ (Seq) **end process**)

$Modify$ (Seq_0 ; wait on L_1 until C_1 ; Seq_1 ; ... wait on L_n until C_n ; Seq_n) $\triangleq$
$\quad$ (**if** pw=1 and $Event(Sensitivity(L_1)$ and C_1
$\quad\quad$ **then** Seq_1 ; pw :=2 ;
$\quad\quad$ **else if** pw=2 and $Event(Sensitivity(L_2)$ and C_2
$\quad\quad\quad$ **then** Seq_2 ; pw :=3 ;
$\quad\quad\quad$ ...
$\quad\quad\quad$ **else if** pw=n and $Event(Sensitivity(L_n)$ and C_n
$\quad\quad\quad\quad$ **then** Seq_n ; Seq_0 ; pw :=1 ; **endif**)

Trans (Conc_1 ; Conc_2) $\triangleq$
$\quad$ $resolved($ **Trans** (Conc_1), **Trans** (Conc_2))$[\widetilde{x}/x]$

TAB. 6.4 – Transformation des instructions concurrentes

$$extract : (\mathcal{V} \cup \mathcal{S}) \times \mathbf{AP} \rightarrow \mathcal{E}xpr$$

$$extract(x,\ y := \mathrm{E}) = \begin{cases} \mathrm{E}, & \text{si } x = y \\ x, & \text{si } x \neq y \text{ et } x \in \mathcal{V} \\ \widetilde{x}, & \text{si } x \neq y \text{ et } x \in \mathcal{S} \end{cases}$$

$$extract(x,\ \mathrm{S} \parallel \mathrm{Q}) = \begin{cases} extract(x, \mathrm{S}), & \text{si } x \in \mathrm{W_S} \\ extract(x, \mathrm{Q}), & \text{autrement} \end{cases}$$

$$substitute : \mathbf{AP} \times \mathbf{AP} \rightarrow \mathbf{AP}$$

$$substitute(x := \mathrm{E},\ y := \mathrm{F}) = y := \mathrm{F}[x/\mathrm{E}] \text{ , si } x \in \mathcal{V}$$
$$substitute(x := \mathrm{E},\ y := \mathrm{F}) = y := \mathrm{F}[\widetilde{x}\ /\ \mathrm{E}] \text{ , si } x \in \mathcal{S}$$
$$substitute(\mathrm{S},\ \mathrm{Q}) = \mathrm{Q} \text{ si } \mathrm{W_S} \cap \mathrm{R_Q} = \emptyset$$
$$substitute(\mathrm{S},\ \mathrm{S}_1 \parallel \mathrm{S}_2 \cdots \parallel \mathrm{S}_n) = \parallel_{i \in 1,n} substitute(\mathrm{S}, \mathrm{S}_i)$$
$$substitute(\mathrm{S}_1 \parallel \mathrm{S}_2 \cdots \parallel \mathrm{S}_n,\ y := \mathrm{F}) := y := \mathrm{F}\ [x/\mathrm{E}, \widetilde{z}\ /\mathrm{G}]$$

$$\forall x \in \cup_{i \in 1,n} \mathrm{W_{S_i}} \cap \mathcal{V}, \mathrm{S}_i : x := \mathrm{E} \text{ et } \forall z \in \cup_{j \in 1,n} \mathrm{W_{S_j}} \cap \mathcal{S}, \mathrm{S}_j : z := \mathrm{G}$$

$$resolved : \mathbf{AP} \times \mathbf{AP} \rightarrow \mathbf{AP}$$

$$resolved(S,T) = \parallel_{x \in W_S \cup W_T - W_S \cap W_T} \{x := E\} \cup \parallel_{x \in W_S \cap W_T \cap \mathcal{S}} \{x := Res(x,E,F), x := E \in S \text{ et } x := F \in T\}$$

TAB. 6.5 – Opérateurs pour la transformation

$$pos : Id \rightarrow Pos$$

$pos(x) = nil$
$pos(id.var) = pos(id)\&(pos_{var}(var, \tau(id)))$
$pos(id(a_1,...,a_n)) = pos(id)\ \&(a_1,...,a_n)$
$pos(id(a_1 \text{ to } a_2)) = pos(id)\ \&\ ((\text{to},\ a_1,\ a_2))$
$pos(id(a_1 \text{ downto } a_2))= pos(id)\ \&\ ((\text{downto},\ a_1,\ a_2))$

τ retourne le type d'un identificateur et pos_{var} retourne la position d'un champ dans un type enregistrement.

$$nth : Pos \times \mathcal{T} \rightarrow \mathcal{T}$$

$nth(p, x) = x$ si $p = nil$
$nth(p, x) = x(p)$ si $p \neq nil$

$$replace : 2^{Pos \times \mathcal{E}xpr} \times \mathcal{T} \rightarrow \mathcal{T}'$$

$replace(L, x) = x$ si $L = nil$
$replace([(p, F), L], x) = F$ si $p = nil$
$replace([(p, F), L], x) = x'$, où

$$x'(k) = \begin{cases} F, & \text{si } k = p \\ F(k), & \text{si } k \in p \\ replace(L, x)(k) & \text{autrement} \end{cases}$$

et $k \in \mathbb{Z}^n$. n est la longueur de la position p (la position étant une liste).

TAB. 6.6 – Des fonctions auxiliaires

Chapitre 7

ANNEXE : La correction de règles de réécriture pour *nth* et *replace*

Théorème 16 *Les règles de simplification pour nth et replace sont correctes.*

Démonstration

Les deux premières règles, a) et b), sont déduites à partir de la définition des fonctions.

Pour la troisième règle, c), il revient à prouver que $nth(p, replace([(p, F), L], x)) = F$.

Nous allons utiliser la définition de *nth*.

Si $p = nil$ alors $nth(p, replace([(p, F), L], x)) = replace([(p, F), L], x)$.

Conformément à la définition de *replace*, si $p = nil$, alors $replace([(p, F), L], x) = F$.

Si $p \neq nil$ alors $nth(p, replace([(p, F), L], x)) = replace([(p, F), L], x)(p)$.

Conformément à la définition de *replace*, ceci se réduit à F.

Pour démontrer la règle d), il faut prouver que, si p et q sont complètement disjointes, alors $nth(p, replace([(q, F), L], x))$ est égal à $nth(p, replace(L, x))$. Conformément à la définition de positions disjointes, $p \neq nil$ et $q \neq nil$. Donc, si nous appliquons la définition de *nth*, $nth(p, replace([(q, F), L], x))$ devient $replace([(q, F), L], x))(p)$. Si nous appliquons la définition de *replace*, l'expression est réduite à :

$$replace([(q, F), L], x))(p) = \begin{cases} F, & \text{si } p = q \\ F(p), & \text{si } p \in q \\ replace(L, x)(p) & \text{autrement} \end{cases}$$

Dans le même temps, la partie droite de l'égalité à prouver $nth(p, replace(L, x))$, se reduit à $replace(L, x))(p)$. Maintenant il faut continuer avec une preuve par cas. Suite à l'hypothèse que p et q sont disjointes et en utilisant la définition des positions, on déduit que $p \neq q$ et $p \notin q$. Donc, l'égalité $replace([(q, F), L], x))(p) = replace(L, x))(p)$ est vraie.

Pour démontrer la règle e), il faut montrer que

$replace(L_1, replace(L_2, x)) = replace(integrate(L_1, L_2), x)$.

Nous allons utiliser l'induction sur la longueur de la liste des positions L_1.

Dans le cas de base, la longueur de la liste est 0. Donc $L_1 = nil$. Conformément à la définition de *replace*, $replace(L_1, replace(L_2, x))$ est égal à $replace(L_2, x)$.

De l'autre part, conformément à la définition de *integrate* :

$replace(integrate(L_1, L_2), x) = replace(L_2, x)$.

Donc l'égalité est prouvée pour le cas de base.

Dans le pas inductif, nous supposons que l'égalité est vraie pour toute liste L, de longueur plus petite ou égale à n (n est un naturel), et nous démontrons qu'elle reste vraie si la longueur de la liste est $n+1$. Donc, la liste L_1 a la forme $[(p, F), Q]$, où Q a la longueur plus petite ou égale à n. Conformément à la définition de *replace* nous avons deux cas :

- si $p = nil$, alors $replace([(p, F), Q], replace(L_2, x))$ devient F. Dans le même temps, $integrate([(p, F), Q], L_2) = [(p, F), Q]\&L'$, où $L' = \{(q, E) \mid (q, E) \in L_2$ et $\forall (r, R) \in [(p, F), Q]$, q et r ne sont pas identiques$\}$. $replace(integrate([(p, F), Q], L_2), x)$ devient donc $replace([(p, F), Q]\&L', x)$, qui se reduit à F, puisque l'opérateur & est commutatif : $[(p, F), Q]\&L' = [(p, F), Q\&L']$.

- si $p \neq nil$, alors

$$replace([(p, F), Q], replace(L_2, x))(k) = \begin{cases} F, \text{ si } k = p \\ F(k), \text{ si } k \in p \\ replace(Q, replace(L_2, x))(k). \end{cases}$$

De l'autre part, $integrate([(p, F), Q], L_2)$ se réduit à $[(p, F), Q]\&L'$ qui, à son tour est égal à $[(p, F), Q\&L']$, où $L' = \{(q, E) \mid (q, E) \in L_2$ et $\forall (r, R) \in [(p, F), Q]$, q et r ne sont pas identiques$\}$. Donc la partie droite de l'egalité est :

$$replace([(p, F), Q\&L'], x)(k) = \begin{cases} F, \text{ si } k = p \\ F(k), \text{ si } k \in p \\ replace(Q\&L', x)(k) \text{ autrement} \end{cases}$$

Nous remarquons pour les deux premiers cas, $k = p$ et $k \in p$, que l'égalité à prouver est vérifiée. Pour tout autre k, nous devons prouver :
$replace(Q, replace(L_2, x))(k) = replace(Q\&L', x)(k)$.

Puisque la longueur de Q est plus petite ou égale à n, nous allons utiliser l'hypothèse d'induction : $replace(Q, replace(L_2, x)))(k) = replace(integrate(Q, L_2), x))(k)$.

Il nous reste donc à montrer :
$replace(Q\&L', x)(k) = replace(integrate(Q, L_2), x)(k)$.

Conformément à sa définition, $integrate(Q, L_2) = Q\&L''$, où $L'' = \{(q, E) \mid (q, E) \in L_2$ et $\forall (r, R) \in Q$, q et r ne sont pas identiques$\}$.

Nous remarquons que $L' \subseteq L''$. Le contraire n'est pas vrai, puisque s'il existe E, tel que $(p, E) \in L_2$, alors $(p, E) \in L''$, mais $(p, E) \notin L'$. Sinon, pour tout autre élément (q, E), où p et q ne sont pas identiques, si $(q, E) \in L''$, alors $(q, E) \in L'$. Donc, s'il n'existe pas E, tel que $(p, E) \in L_2$, l'égalité est démontrée.

S'il existe E, tel que $(p, E) \in L_2$, alors L'' est de la forme $A\&[(p, E), B]$, et L' est de forme $A\&B$. Aussi, $\forall (r, R) \in A, ou \in B$, r et p ne sont pas identiques. L'égalité devient

$replace(Q\&A\&[(p, E), B], x)(k) = replace(Q\&A\&B, x)(k)$.

Puisque nous devons prouver l'égalité pour tout élément qui se trouve sur la position k, où k a la même longueur que p et k et p ne sont pas identiques, l'égalité est démontrée.

QED

Chapitre 8

ANNEXE : Sortie du démonstrateur ACL2

Sortie du démonstrateur ACL2 pour la définition de la fonction fib.

```
ACL2 !>(defun fib (n)
  (if (zp n) n
    (if (equal n 1) n
      (+ (fib (- n 1)) (fib (- n 2))))))


For the admission of FIB we will use the relation O< (which is known
to be well-founded on the domain recognized by O-P) and the measure
(ACL2-COUNT N).  The non-trivial part of the measure conjecture is


Goal
(IMPLIES (AND (NOT (ZP N)) (NOT (EQUAL N 1)))
         (O< (ACL2-COUNT (+ -2 N))
             (ACL2-COUNT N))).


This simplifies, using the :compound-recognizer rule
ZP-COMPOUND-RECOGNIZER,the :definitions ACL2-COUNT, INTEGER-ABS, O-FINP
and O< and primitive type reasoning, to the following two conjectures.


Subgoal 2
(IMPLIES (AND (NOT (ZP N))
              (NOT (EQUAL N 1))
              (< (+ -2 N) 0))
         (< (- (+ -2 N)) N)).


But simplification reduces this to T, using the :compound-recognizer
```

```
rule ZP-COMPOUND-RECOGNIZER, linear arithmetic
and primitive type reasoning.


Subgoal 1
(IMPLIES (AND (NOT (ZP N))
              (NOT (EQUAL N 1))
              (<= 0 (+ -2 N)))
         (< (+ -2 N) N)).


But simplification reduces this to T, using linear arithmetic.


Q.E.D.


That completes the proof of the measure theorem for FIB.  Thus, we
admit this function under the principle of definition. We could deduce
no constraints on the type of FIB.


Summary
Form :  ( DEFUN FIB ...)
Rules : (( :COMPOUND-RECOGNIZER ZP-COMPOUND-RECOGNIZER)
        ( :DEFINITION ACL2-COUNT)
        ( :DEFINITION INTEGER-ABS)
        ( :DEFINITION O-FINP)
        ( :DEFINITION O<)
        ( :FAKE-RUNE-FOR-LINEAR NIL)
        ( :FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings :  None
Time :  0.01 seconds (prove : 0.00, print : 0.00, other : 0.01)
 FIB
```

Sortie du démonstrateur ACL2 pour la définition de la fonction *gcdD* sans déclaration de la mesure.

```
ACL2 !>(defun gcdD (a b)
  (cond ((zp a) b)
        ((zp b) a)
        ((equal a b) a)
        (t  (if (< a b)
```

```
               (gcdD a (- b a))
              (gcdD (- a b) b)))))

ACL2 Error in ( DEFUN GCDD ...) :  No :MEASURE was supplied with the
definition of GCDD.  Our heuristics for guessing one have not made
any suggestions.  No argument of the function is tested along every
branch and occurs as a proper subterm at the same argument position
in every recursive call.  You must specify a :MEASURE.  See :DOC defun.


Summary
Form :  ( DEFUN GCDD ...)
Rules : NIL
Warnings :  None
Time :  0.00 seconds (prove : 0.00, print : 0.00, other : 0.00)
******** FAILED ********  See :DOC failure  ******** FAILED ********
```

Sortie du démonstrateur ACL2 pour la définition de la fonction *gcdD* avec déclaration de la mesure.

```
ACL2 !>(defun gcdD (a b)
 (declare (xargs :measure (acl2-count (+ a b))))
   (cond ((zp a) b)
         ((zp b) a)
         ((equal a b) a)
         (t  (if (< a b)
                 (gcdD a (- b a))
               (gcdD (- a b) b)))))


For the admission of GCDD we will use the relation O< (which is known
to be well-founded on the domain recognized by O-P) and the measure
(ACL2-COUNT (+ A B)).  The non-trivial part of the measure conjecture
is


Goal
(AND (IMPLIES (AND (NOT (ZP A))
                   (NOT (ZP B))
                   (NOT (EQUAL A B))
                   (<= B A))
              (O< (ACL2-COUNT (+ (+ A (- B)) B))
```

```
                (ACL2-COUNT (+ A B))))
     (IMPLIES (AND (NOT (ZP A))
                   (NOT (ZP B))
                   (NOT (EQUAL A B))
                   (< A B))
              (O< (ACL2-COUNT (+ A B (- A)))
                  (ACL2-COUNT (+ A B))))).
```

By the simple :rewrite rule ASSOCIATIVITY-OF-+ we reduce the conjecture to the following two conjectures.

```
Subgoal 2
(IMPLIES (AND (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (<= B A))
         (O< (ACL2-COUNT (+ A (- B) B))
             (ACL2-COUNT (+ A B)))).
```

This simplifies, using the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER, the :definitions ACL2-COUNT, FIX, INTEGER-ABS, O-FINP and O<, primitive type reasoning and the :rewrite rules COMMUTATIVITY-OF-+, INVERSE-OF-+ and UNICITY-OF-0, to

```
Subgoal 2'
(IMPLIES (AND (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (<= B A))
         (< A (+ A B))).
```

But simplification reduces this to T, using the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER and linear arithmetic.

```
Subgoal 1
(IMPLIES (AND (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
```

```
             (< A B))
         (O< (ACL2-COUNT (+ A B (- A)))
             (ACL2-COUNT (+ A B)))).
```

This simplifies, using the :compound-recognizer rule
ZP-COMPOUND-RECOGNIZER,the :definitions ACL2-COUNT, INTEGER-ABS, O-FINP
and O<, primitive type reasoning and the :rewrite rule COMMUTATIVITY-OF-+,
to the following two conjectures.

```
Subgoal 1.2
(IMPLIES (AND (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (< A B)
              (< (+ A (- A) B) 0))
         (< (- (+ A (- A) B)) (+ A B))).
```

But simplification reduces this to T, using the :compound-recognizer
rule ZP-COMPOUND-RECOGNIZER, linear arithmetic
and primitive type reasoning.

```
Subgoal 1.1
(IMPLIES (AND (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (< A B)
              (<= 0 (+ A (- A) B)))
         (< (+ A (- A) B) (+ A B))).
```

But simplification reduces this to T, using the :compound-recognizer
rule ZP-COMPOUND-RECOGNIZER and linear arithmetic.

```
Q.E.D.
```

That completes the proof of the measure theorem for GCDD. Thus, we
admit this function under the principle of definition. We observe that
the type of GCDD is described by the theorem
(OR (INTEGERP (GCDD A B)) (EQUAL (GCDD A B) B)). We used the :compound-

```
recognizer rule ZP-COMPOUND-RECOGNIZER and primitive type reasoning.


Summary
Form :  ( DEFUN GCDD ...)
Rules : (( :COMPOUND-RECOGNIZER ZP-COMPOUND-RECOGNIZER)
        ( :DEFINITION ACL2-COUNT)
        ( :DEFINITION FIX)
        ( :DEFINITION INTEGER-ABS)
        ( :DEFINITION NOT)
        ( :DEFINITION O-FINP)
        ( :DEFINITION O<)
        ( :FAKE-RUNE-FOR-LINEAR NIL)
        ( :FAKE-RUNE-FOR-TYPE-SET NIL)
        ( :REWRITE ASSOCIATIVITY-OF-+)
        ( :REWRITE COMMUTATIVITY-OF-+)
        ( :REWRITE INVERSE-OF-+)
        ( :REWRITE UNICITY-OF-0))
Warnings :  None
Time :  0.02 seconds (prove : 0.00, print : 0.02, other : 0.00)
 GCDD
```

Sortie du démonstrateur ACL2 pour la définition de la fonction *gcdD* avec déclaration de la mesure et des gardes.

```
ACL2 !>(defun gcdD-gardes (a b)
 (declare (xargs :measure (acl2-count (+ a b))
                 :guard (and (integerp a) (<= 0 a)
                             (integerp b) (<= 0 b))))
   (cond ((zp a) b)
         ((zp b) a)
         ((equal a b) a)
         (t  (if (< a b)
                 (gcdD-gardes a (- b a))
               (gcdD-gardes (- a b) b)))))


For the admission of GCDD-GARDES we will use the relation O< (which
is known to be well-founded on the domain recognized by O-P) and the
measure (ACL2-COUNT (+ A B)).  The non-trivial part of the measure
conjecture is
```

```
Goal
(AND (IMPLIES (AND (NOT (ZP A))
                   (NOT (ZP B))
                   (NOT (EQUAL A B))
                   (<= B A))
              (O< (ACL2-COUNT (+ (+ A (- B)) B))
                  (ACL2-COUNT (+ A B))))
     (IMPLIES (AND (NOT (ZP A))
                   (NOT (ZP B))
                   (NOT (EQUAL A B))
                   (< A B))
              (O< (ACL2-COUNT (+ A B (- A)))
                  (ACL2-COUNT (+ A B))))).
```

By the simple :rewrite rule ASSOCIATIVITY-OF-+ we reduce the conjecture to the following two conjectures.

```
Subgoal 2
(IMPLIES (AND (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (<= B A))
         (O< (ACL2-COUNT (+ A (- B) B))
             (ACL2-COUNT (+ A B)))).
```

This simplifies, using the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER, the :definitions ACL2-COUNT, FIX, INTEGER-ABS, O-FINP and O<, primitive type reasoning and the :rewrite rules COMMUTATIVITY-OF-+, INVERSE-OF-+ and UNICITY-OF-0, to

```
Subgoal 2'
(IMPLIES (AND (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (<= B A))
         (< A (+ A B))).
```

But simplification reduces this to T, using the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER and linear arithmetic.

```
Subgoal 1
(IMPLIES (AND (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (< A B))
         (O< (ACL2-COUNT (+ A B (- A)))
             (ACL2-COUNT (+ A B)))).
```

This simplifies, using the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER, the :definitions ACL2-COUNT, INTEGER-ABS, O-FINP and O<, primitive type reasoning and the :rewrite rule COMMUTATIVITY-OF-+, to the following two conjectures.

```
Subgoal 1.2
(IMPLIES (AND (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (< A B)
              (< (+ A (- A) B) 0))
         (< (- (+ A (- A) B)) (+ A B))).
```

But simplification reduces this to T, using the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER, linear arithmetic and primitive type reasoning.

```
Subgoal 1.1
(IMPLIES (AND (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (< A B)
              (<= 0 (+ A (- A) B)))
         (< (+ A (- A) B) (+ A B))).
```

But simplification reduces this to T, using the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER and linear arithmetic.

Q.E.D.

```
That completes the proof of the measure theorem for GCDD-GARDES.  Thus,
we admit this function under the principle of definition. We observe
that the type of GCDD-GARDES is described by the theorem
(OR (INTEGERP (GCDD-GARDES A B)) (EQUAL (GCDD-GARDES A B) B)).  We
used the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER and primitive
type reasoning.


The non-trivial part of the guard conjecture for GCDD-GARDES, given
the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER and primitive
type reasoning, is


Goal
(AND (IMPLIES (AND (<= 0 B)
                   (INTEGERP B)
                   (<= 0 A)
                   (INTEGERP A)
                   (NOT (ZP A))
                   (NOT (ZP B))
                   (NOT (EQUAL A B))
                   (< A B))
              (<= 0 (+ B (- A))))
     (IMPLIES (AND (<= 0 B)
                   (INTEGERP B)
                   (<= 0 A)
                   (INTEGERP A)
                   (NOT (ZP A))
                   (NOT (ZP B))
                   (NOT (EQUAL A B))
                   (<= B A))
              (<= 0 (+ A (- B))))).


By case analysis we reduce the conjecture to the following two conjectures.


Subgoal 2
(IMPLIES (AND (<= 0 B)
              (INTEGERP B)
              (<= 0 A)
              (INTEGERP A)
```

```
              (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (< A B))
         (<= 0 (+ B (- A)))).
```

But simplification reduces this to T, using the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER, linear arithmetic and primitive type reasoning.

```
Subgoal 1
(IMPLIES (AND (<= 0 B)
              (INTEGERP B)
              (<= 0 A)
              (INTEGERP A)
              (NOT (ZP A))
              (NOT (ZP B))
              (NOT (EQUAL A B))
              (<= B A))
         (<= 0 (+ A (- B)))).
```

But simplification reduces this to T, using the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER, linear arithmetic and primitive type reasoning.

Q.E.D.

That completes the proof of the guard theorem for GCDD-GARDES. GCDD-GARDES is compliant with Common Lisp.

```
Summary
Form :  ( DEFUN GCDD-GARDES ...)
Rules : (( :COMPOUND-RECOGNIZER ZP-COMPOUND-RECOGNIZER)
        ( :DEFINITION ACL2-COUNT)
        ( :DEFINITION FIX)
        ( :DEFINITION INTEGER-ABS)
        ( :DEFINITION NOT)
        ( :DEFINITION O-FINP)
```

```
        ( :DEFINITION O<)
        ( :FAKE-RUNE-FOR-LINEAR NIL)
        ( :FAKE-RUNE-FOR-TYPE-SET NIL)
        ( :REWRITE ASSOCIATIVITY-OF-+)
        ( :REWRITE COMMUTATIVITY-OF-+)
        ( :REWRITE INVERSE-OF-+)
        ( :REWRITE UNICITY-OF-0))
Warnings :  None
Time :  0.06 seconds (prove : 0.00, print : 0.06, other : 0.00)
 GCDD-GARDES
```

Chapitre 9

ANNEXE : La spécification ACL2 de l'algorithme SHA-1

```
(IN-PACKAGE "ACL2")


(defun Ch (x y z)
  (b-xor (b-and x y) (b-and (b-not x) z)))


(defun Parity (x y z)
   (b-xor  x y z))


(defun Maj (x y z)
  (b-xor (b-and x y) (b-and x z) (b-and y z)))


(defun F (i x y z)
  (cond ((and (<= 0 i) (<= i 19))
 (Ch x y z))
((or (and (<= 20 i) (<= i 39)) (and (<= 60 i) (<= i 79)))
 (Parity x y z))
((and (<= 40 i) (<= i 59))
 (Maj x y z))
(t nil)))


(defun sigma-0-256 (x)
  (b-xor  (rotr 2 x) (rotr 13 x) (rotr 22 x)))


(defun sigma-1-256 (x)
```

```
  (b-xor (rotr 6 x) (rotr 11 x) (rotr 25 x)))


(defun s-0-256 (x)
  (b-xor (rotr 7 x) (rotr 18 x) (shr 3 x)))


(defun s-1-256 (x)
  (b-xor (rotr 17 x) (rotr 19 x) (shr 10 x)))


(defun sigma-0-512 (x)
  (b-xor  (rotr 28 x) (rotr 34 x) (rotr 39 x)))


(defun sigma-1-512 (x)
  (b-xor  (rotr 14 x) (rotr 18 x) (rotr 41 x)))


(defun s-0-512 (x)
  (b-xor  (rotr 1 x) (rotr 8 x) (shr 7 x)))


(defun s-1-512 (x)
  (b-xor  (rotr 19 x) (rotr 61 x) (shr 6 x)))


(defun padding-1-256 (m)
  (if (<= (mod (1+ (len m)) 512) 448)
      (append m (list 1)
      (make-list (- 448 (mod (1+ (len m)) 512))
 :initial-element 0 )
      (bv-to-n (int-bv-be (len m)) 64))
    (append m (list 1)
    (make-list (- 960 (mod (1+ (len m)) 512))
       :initial-element 0 )
    (bv-to-n (int-bv-be (len m)) 64))))


(defun parsing (m n)
  (declare (xargs :measure (len m)))
  (cond ((endp m) nil)
((zp n) nil)
(t (cons (firstn n m) (parsing (nthcdr n m) n)))))
;ACL2 !>(parsing '(0 1 2 3 4 5 6 7) 3)
```

```
;((0 1 2) (3 4 5) (6 7))
; constants of sha-1


(defun K (i)
  (cond ((and (<= 0 i) (<= i 19))
 '(0 1 0 1 1 0 1 0 1 0 0 0
     0 0 1 0 0 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1))
((and (<= 20 i) (<= i 39))
 '(0 1 1 0 1 1 1 0 1 1 0 1
     1 0 0 1 1 1 1 0 1 0 1 1 1 0 1 0 0 0 0 1))
((and (<= 40 i) (<= i 59))
 '(1 0 0 0 1 1 1 1 0 0 0 1
     1 0 1 1 1 0 1 1 1 1 0 0 1 1 0 1 1 1 0 0))
((and (<= 60 i) (<= i 79))
 '(1 1 0 0 1 0 1 0 0 1 1 0
     0 0 1 0 1 1 0 0 0 0 0 1 1 1 0 1 0 1 1 0))
(t nil)))
; initial hash values for sha-1


(defconst *h0*
  '(0 1 1 0 0 1 1 1 0 1 0 0
      0 1 0 1 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1))


(defconst *h1*
  '(1 1 1 0 1 1 1 1 1 1 0 0
      1 1 0 1 1 0 1 0 1 0 1 1 1 0 0 0 1 0 0 1))


(defconst *h2*
   '(1 0 0 1 1 0 0 0 1 0 1 1
       1 0 1 0 1 1 0 1 1 1 0 0 1 1 1 1 1 1 1 0))


(defconst *h3*
   '(0 0 0 1 0 0 0 0 0 0 1 1
       0 0 1 0 0 1 0 1 0 1 0 0 0 1 1 1 0 1 1 0))


(defconst *h4*
   '(1 1 0 0 0 0 1 1 1 1 0 1
       0 0 1 0 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0 0))
```

```
;constant of sha-1


(defconst *mask*
  '(0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1))
;--sha-1


(defun s (j)
  (bv-int-be (b-and (int-bv-be j) *mask*)))


(defun temp-spec (j working-variables m-i)
  (plus (rotl 5 (nth 0 working-variables))
(F j (nth 1 working-variables)
    (nth 2 working-variables)
    (nth 3 working-variables))
(nth 4 working-variables)
(nth (s j) m-i)
(K j)))


(defun word-spec (j  m-i)
  (rotl 1 (b-xor
   (nth (bv-int-be (b-and (int-bv-be (+ 13 (s j)))
  *mask*))  m-i)
   (nth (bv-int-be (b-and (int-bv-be (+ 8 (s j)))
  *mask*))  m-i)
   (nth (bv-int-be (b-and (int-bv-be (+ 2 (s j)))
  *mask*))  m-i)
   (nth (bv-int-be (b-and (int-bv-be j)
  *mask*))  m-i))))


(defun digest-one-block-spec (j working-variables  m-i)
  (declare (xargs :measure (acl2-count (- 80 j))))
  (if (and (integerp j) (<= 0 j))
      (cond ((<= 80 j) working-variables)
    (t (digest-one-block-spec
(+ 1 j)
(list (temp-spec j working-variables
 (if (<= 16 j)
     (repl (s j) (word-spec j m-i) m-i)
```

```
   m-i))
      (nth 0 working-variables)
      (rotl 30 (nth 1 working-variables))
      (nth 2 working-variables)
      (nth 3 working-variables))
       (if (<= 16 j) (repl (s j) (word-spec j m-i) m-i) m-i))))
   nil))


(defun intermediate-hash (l1 l2)
  (list (plus (nth 0 l1) (nth 0 l2))
(plus (nth 1 l1) (nth 1 l2))
(plus (nth 2 l1) (nth 2 l2))
(plus (nth 3 l1) (nth 3 l2))
(plus (nth 4 l1) (nth 4 l2))))


(defun digest-spec (m hash-values)
  (if (endp  m) hash-values
    (digest-spec (cdr m)
 (intermediate-hash
  hash-values
  (digest-one-block-spec 0 hash-values
 (parsing (car m) 32))))))


(defun sha-norm (m)
  (digest-spec (parsing (padding-1-256 m) 512)
       (list *h0* *h1* *h2* *h3* *h4*)))
```

Chapitre 10

ANNEXE : Le code VHDL du SHA-1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

package sha_function is

function ch(e,f,g : in std_logic_vector(31 downto 0))
         return std_logic_vector;
function Maj(a,b,c : in std_logic_vector(31 downto 0))
         return std_logic_vector;
function add2(h,e,ch1,a,maj1,wi32,ki32 : in std_logic_vector(31 downto 0))
         return std_logic_vector;
function shift(X : in std_logic_vector(31 downto 0))
         return std_logic_vector;
function ajout2(X,Y : in std_logic_vector(31 downto 0))
         return std_logic_vector;
function adressage2(count : in std_logic_vector(3 downto 0);t :in integer)
         return std_logic_vector;

constant idle : std_logic_vector(2 downto 0) : ="000";
constant init : std_logic_vector(2 downto 0) : ="001";
constant sha_ini_one : std_logic_vector(2 downto 0) : ="010";
constant calculW_one : std_logic_vector(2 downto 0) : ="011";
constant calcul_abc_one : std_logic_vector(2 downto 0) : ="100";
constant result : std_logic_vector(2 downto 0) : ="101";
constant resultW : std_logic_vector(2 downto 0) : ="110";
end;
```

```
package body sha_function is

function ch(e,f,g : in std_logic_vector(31 downto 0))
            return std_logic_vector is
   variable ch1 : std_logic_vector(31 downto 0);
   begin
      ch1 : =(e and f)xor(not(e)and g);
   return ch1;
end ch;

function Maj(a,b,c : in std_logic_vector(31 downto 0))
            return std_logic_vector is
   variable Maj1 : std_logic_vector(31 downto 0);
   begin
      Maj1 : =(a and b) xor (a and c) xor (b and c);
   return Maj1;
end Maj;

function add2(h,e,ch1,a,maj1,wi32,ki32 : in std_logic_vector(31 downto 0))
            return std_logic_vector is
   variable s : std_logic_vector(31 downto 0);
   variable sig_2 : std_logic_vector(31 downto 0);
   variable sig : std_logic_vector(31 downto 0);
   begin
      sig_2 : =(e(5 downto 0) & e(31 downto 6) xor
               e(10 downto 0) & e(31 downto 11) xor
               e(24 downto 0) & e(31 downto 25));
      sig : =(a(1 downto 0) & a(31 downto 2) xor
            a(12 downto 0) & a(31 downto 13) xor
            a(21 downto 0) & a(31 downto 22));
      s : =h +sig_2  + Ch1 + sig + Maj1+ ki32+ wi32;
   return s;
end add2;

function shift(X : in std_logic_vector(31 downto 0))
            return std_logic_vector is
   variable Y : std_logic_vector(31 downto 0);
   begin
      Y : =X(30 downto 0) &X(31);
   return Y;
end shift;
```

```
function ajout2(X,Y : in std_logic_vector(31 downto 0))
            return std_logic_vector is
   variable Z : std_logic_vector(31 downto 0) ;
   begin
      Z : =X+Y ;
   return Z ;
end ajout2 ;

function adressage2(count : in std_logic_vector(3 downto 0) ;t : in integer)
            return std_logic_vector is
   variable X : std_logic_vector(3 downto 0) ;
   begin
   X : =count ;
   if  t=18 then X : =count ;
    elsif t=19  then  X : =X+2 ;
    elsif t=20  then  X : =X+8 ;
    elsif t=21  then  X : =X+13 ;
   end if ;
   return X ;
end adressage2 ;
end sha_function ;

entity sha_algorithm is
port (clk : in std_logic ;
      k : out std_logic_vector(6 downto 0) ;
      cnt : out std_logic ;
      etat : in std_logic_vector(2 downto 0) ;
      rdata : in std_logic_vector(31 downto 0) ;
      wdata : out std_logic_vector(31 downto 0) ;
      base_addr : in STD_LOGIC_VECTOR(11 downto 0) ;
      addr : out STD_LOGIC_VECTOR(11 downto 0) ;
      aout,bout,cout,dout,eout : out std_logic_vector(31 downto 0) ;
      l_bloc : in std_logic_vector(5 downto 0)
      ) ;
end sha_algorithm ;

architecture archi3 of sha_algorithm is

      type word32 is  array(0 to 63) of std_logic_vector(31 downto 0) ;
      signal count : std_logic_vector(7 downto 0) ;
      signal a,b,c,d,e,wi32 : std_logic_vector(31 downto 0) ;
      signal a_mem,b_mem,c_mem,d_mem,e_mem : std_logic_vector(31 downto 0) ;
```

```
      signal t : std_logic_vector(5 downto 0);

begin
-------------------gestion compteur-------------------
process(clk)
begin
      if clk'event and clk='1' then
         if etat/="000" then
              t<=t+1;
              if etat="100" and CONV_INTEGER(t)=22 then t<="010010";
                 elsif etat="001" and CONV_INTEGER(l_bloc)/=0 then
                       t<="000010";
              end if;
            else t<="000000";
         end if;
      end if;
end process;

-------------------gestion des adresses-------------------
with etat select
  addr<= base_addr+count(3 downto 0)+CONV_INTEGER(l_bloc)*16 when "010",
         base_addr+adressage2(count(3 downto 0),CONV_INTEGER(t))+
             CONV_INTEGER(l_bloc)*16 when "011",
         base_addr+count(3 downto 0)+CONV_INTEGER(l_bloc)*16 when "100",
         base_addr when others;

-------------------gestion des ecritures-------------------
with conv_integer(t) select
      wdata<=wi32 when 22,-------ecriture du W calculé au cycle 22
      x"00000000" when others;

process(t,etat,clk)
begin
      if clk'event and clk='1' then
         case etat is
              when idle => count<="00000000";
                           cnt<='0';
-------------------init ( etat 1)-------------------
              when init => if CONV_INTEGER(t)>=0 then
                              if l_bloc="000000" then
                                   a<=x"67452301";
                                   b<=x"efcdab89";
```

```
                                c<=x"98badcfe";
                                d<=x"10325476";
                                e<=x"c3d2e1f0";
                            end if;
                            cnt<='1';
                        end if;

-------------------sha_ini_one ( etat 2)-------------------
            when sha_ini_one=> wi32<=rdata;
                                e<=d;
                                d<=c;
                                c<=b(1 downto 0) & b(31 downto 2);
                                b<=a;
                                a<=ch(b,c,d)+e+rdata+(A(26 downto 0) &
                                   A(31 downto 27))+x"5A827999";
                                count<=count+1;
                                cnt<='0';

-------------------calculW_one(etat 3)--------------------
            when calculW_one=> if CONV_INTEGER(t)=18 then
                                   wi32<=rdata;
                               elsif CONV_INTEGER(t)=19 then
                                   wi32<=wi32 xor rdata;
                               elsif CONV_INTEGER(t)=20 then
                                   wi32<=wi32 xor rdata;
                                   cnt<='1';
                               elsif CONV_INTEGER(t)=21 then
                                   wi32<= Shift(wi32 xor rdata);
                                   cnt<='0';
                               end if;

-------------------calcul_abc_one(etat 4) --------------------
            when calcul_abc_one=>
                   if CONV_INTEGER(t)=22 then
                      e<=d;
                      d<=c;
                      c<=b(1 downto 0) & b(31 downto 2);
                      b<=a;
                      if CONV_INTEGER(count)<=19 then
                         a<=ajout2(ch(b,c,d),ajout2(e,wi32))+
                            (A(26 downto 0) & A(31 downto 27))
                            + x"5A827999";
```

```
                        elsif (CONV_INTEGER(count)>=20 and
                               CONV_INTEGER(count)<=39)
                           then a<=(b xor c xor d)+ajout2(e,wi32)+
                                    (A(26 downto 0) & A(31 downto 27))
                                     + x"6ED9EBA1";
                        elsif (CONV_INTEGER(count)>=40 and
                               CONV_INTEGER(count)<=59)
                           then a<=ajout2(maj(b,c,d),ajout2(e,wi32))+
                                    (A(26 downto 0) & A(31 downto 27))
                                    + x"8F1BBCDC";
                        else a<=(b xor c xor d)+ajout2(e,wi32)+
                                 (A(26 downto 0) & A(31 downto 27))
                                  + x"ca62C1D6";
                        end if;
                        count<=count+1;
                     end if;

------------------- result (etat 5) --------------------
               when result=>
                     if CONV_INTEGER(t)=18 then
                        if l_bloc="000001" then-----test premier bloc-------
                        a<=ajout2(a,x"67452301");---utilise les hi initiaux
                        b<=ajout2(b,x"efcdab89");
                        c<=ajout2(c,x"98badcfe");
                        d<=ajout2(d,x"10325476");
                        e<=ajout2(e,x"c3d2e1f0");
                     else ----------------autre bloc----------
                        a<=ajout2(a,a_mem);----utilise les hi mis en mémoire
                        b<=ajout2(b,b_mem);
                        c<=ajout2(c,c_mem);
                        d<=ajout2(d,d_mem);
                        e<=ajout2(e,e_mem);
                     end if;
                     end if;
                     if CONV_INTEGER(t)=19 then
                        aout<=a;
                        bout<=b;
                        cout<=c;
                        dout<=d;
                        eout<=e;
                        a_mem<=a;
                        b_mem<=b;
```

```
                        c_mem<=c ;
                        d_mem<=d ;
                        e_mem<=e ;
                        cnt<='1' ;
                    end if ;

-------------------ecrit result (etat 6) --------------------
              when resultw=> cnt<='0' ;
                             count<="00000000";-remise à zero des compteurs
              when others =>null ;
          end case ;
      end if ;
end process ;
k<=count(6 downto 0) ;
end archi3 ;

entity sha_fsm is port(
    Reset : in std_logic ;
    ClK : in std_logic ;
    start : in std_logic ;
    CNT : in std_logic ;
    k : in std_logic_vector(6 downto 0) ;
    etatout : out std_logic_vector(2 downto 0) ;
    ram_sel : out std_logic ;
    busy : out std_logic ;
    ram_write : out std_logic ;
    l_bloc : out std_logic_vector(5 downto 0) ;
    nb_bloc : in std_logic_vector(5 downto 0) ;
    reset_done : in std_logic ;
    done : out std_logic) ;
end sha_fsm ;

architecture fsm of sha_fsm is

      signal etat : std_logic_vector(2 downto 0) ;
      signal bl : std_logic_vector(5 downto 0) ;

begin
etatout<=etat ;
l_bloc<=nb_bloc-bl ;

regetat : process(ClK, Reset,nb_bloc)
```

```
begin
      if Reset='1' then
         etat<=idle;
         done<='0';
      elsif ClK'event and clk='1' then
         case etat is
            when idle=>
                 if start='1' then
                    etat<=init;
                    bl<=nb_bloc;
                    done<='0';
                 elsif reset_done='1' then
                    done<='0';
                 end if;

            when init=>
                 if cnt='1' or bl/=nb_bloc then
                    etat<=sha_ini_one;
                 else
                    etat<=init;
                 end if;

            when sha_ini_one=>
                 if conv_integer(k)=15 then
                    etat<=calculW_one;
                 else
                    etat<=sha_ini_one;
                 end if;

            when calculW_one=>
                 if cnt = '1' then
                    etat<= calcul_abc_one;
                 else
                    etat<=calculW_one;
                 end if;

            when calcul_abc_one =>
                 if conv_integer(k)=79 then
                    bl<=bl-1;
                    etat <= result;
                 elsif conv_integer(k)<=79 and cnt='1' then
                    etat<=calcul_abc_one;
```

```
                elsif conv_integer(k)<=79 and cnt='0' then
                etat<=calculW_one;
             end if;

         when result =>
             if CNT='1' then etat<=resultw;
             else etat<=result;
             end if;

         when resultW =>
             if CNT='0' then
                if bl="000000" then
                   done<='1';
                   etat<=idle;
                else etat<=init;
                end if;
             else etat<=ResultW;
             end if;

         when others=> end case;
    end if;
end process;

genq2 : process(etat,reset_done)
begin
    if etat=idle then busy<='0';
    else busy<='1';
    end if;
    case etat is
        when idle=>
            ram_sel<='0';
            ram_write<='0';

        when init=>
            ram_sel<='1';
            ram_write<='0';

        when calculW_one=>
            ram_sel<='1';
            ram_write<='0';

        when calcul_abc_one =>
```

```
                ram_sel<='1';
                ram_write<='1';

           when result =>
                ram_sel<='1';
                ram_write<='0';

           when resultW =>
                ram_sel<='1';
                ram_write<='1';

           when others=>
                ram_sel<='1';
                ram_write<='0';
      end case;
end process;
end fsm;

entity sha_top is port (
   start,reset_done : in std_logic;
   ClK, Reset : in std_logic;
   rdata : in std_logic_vector(31 downto 0);
   base_addr : in std_logic_vector (11 downto 0);
   addr : out std_logic_vector(11 downto 0);
   wdata : out std_logic_vector(31 downto 0);
   ram_sel, ram_write : out std_logic;
   busy,done : out std_logic;
   nb_bloc : in std_logic_vector(5 downto 0));
end sha_top;

architecture archi of sha_top is

component sha_fsm port(
   Reset : in std_logic;
   ClK : in std_logic;
   start : in std_logic;
   CNT : in std_logic;
   k : in std_logic_vector(6 downto 0);
   etatout : out std_logic_vector(2 downto 0);
   ram_sel : out std_logic;
   busy : out std_logic;
   ram_write : out std_logic;
```

```
   nb_bloc : in std_logic_vector(5 downto 0);
   l_bloc : out std_logic_vector(5 downto 0);
   reset_done : in std_logic;
   done : out std_logic);
end component;

component sha_algorithm port(
   clk : in std_logic;
   k : out std_logic_vector(6 downto 0);
   cnt : out std_logic;
   etat : in std_logic_vector(2 downto 0);
   rdata : in std_logic_vector(31 downto 0);
   wdata : out std_logic_vector(31 downto 0);
   base_addr : in STD_LOGIC_VECTOR(11 downto 0);
   addr : out STD_LOGIC_VECTOR(11 downto 0);
   aout,bout,cout,dout,eout : out std_logic_vector(31 downto 0);
   l_bloc : in std_logic_vector(5 downto 0));
end component;

   signal k : std_logic_vector(6 downto 0);
   signal cnt : std_logic;
   signal etat : std_logic_vector(2 downto 0);
   signal l_bloc : std_logic_vector(5 downto 0);
   signal aout,bout,cout,dout,eout : std_logic_vector(31 downto 0);

begin

m1 : sha_fsm port map(
   reset=>reset, clk=>clk, start=>start, CNT=>cnt,
   k=>k, etatout=>etat, ram_sel=>ram_sel, busy=>busy,
   ram_write=>ram_write, nb_bloc=>nb_bloc, l_bloc=>l_bloc,
   reset_done=>reset_done, done=>done);

cal_sha : sha_algorithm port map(
   clk=>clk, k=>k, cnt=>cnt, etat=>etat,
   rdata=>rdata, wdata=>wdata, base_addr=>base_addr, addr=>addr,
   l_bloc=>l_bloc, aout=>aout, bout=>bout, cout=>cout,
   dout=>dout, eout=>eout);

end archi;
```

Chapitre 11

ANNEXE : La modélisation en ACL2 du SHA-1 RTL

```
(in-package "ACL2")
(include-book "func")
(defconst *idle* (list 0 0 0))
(defconst *init* (list 0 0 1))
(defconst *sha_ini_one* (list 0 1 0))
(defconst *calculw_one* (list 0 1 1))
(defconst *calcul_abc_one* (list 1 0 0))
(defconst *result* (list 1 0 1))
(defconst *resultw* (list 1 1 0))


(defun nextsig_etat (reset start cnt bl nb_bloc etat k)
  (if (equal reset 1) *idle*
    (if (equal etat *idle*)
        (if (equal start 1) *init* etat)
      (if (equal etat *init*)
          (if (or (equal cnt 1) (not (equal bl nb_bloc)))
              *sha_ini_one*
            etat)
        (if (equal etat *sha_ini_one*)
            (if (equal (conv_integer k) 15)
                *calculw_one*
              etat)
          (if (equal etat *calculw_one*)
              (if (equal cnt 1) *calcul_abc_one* etat)
            (if (equal etat *calcul_abc_one*)
                (if (equal (conv_integer k) 79)
                    *result*
```

```
                (if (and (<= (conv_integer k) 79) (equal cnt 1))
                    etat
                  (if (and (<= (conv_integer k) 79) (equal cnt 0))
                      *calculw_one*
                    etat)))
              (if (equal etat *result*)
                  (if (equal cnt 1) *resultw* etat)
                (if (equal etat *resultw*)
                    (if (equal cnt 0)
                        (if (equal bl (list 0 0 0 0 0 0))
                            *idle*
                          *init*)
                      etat)
                  etat)))))))))


(defun nextsig_etatout (reset start cnt bl nb_bloc etat k)
  (nextsig_etat reset start cnt bl nb_bloc etat k))


(defun nextsig_done (reset reset_done start etat cnt bl done)
  (if (equal reset 1) 0
    (if (equal etat *idle*)
        (if (or (equal start 1) (equal reset_done 1)) 0 done)
      (if (and (equal etat *resultw*)
               (equal cnt 0)
               (equal bl (list 0 0 0 0 0 0)))
          1 done))))


(defun nextsig_bl (reset start etat k nb_bloc bl)
  (if (equal reset 0)
      (if (equal etat *idle*)
          (if (equal start 1) nb_bloc bl)
        (if (and (equal etat *calcul_abc_one*)
                 (equal (conv_integer k) 79))
            (minus bl 1) bl))
    bl))


(defun nextsig_l_bloc ( reset start etat k nb_bloc bl)
  (minus nb_bloc  (nextsig_bl reset start etat k nb_bloc bl)))
```

```
(defun nextsig_busy (reset start cnt bl nb_bloc etat k)
  (if (equal (nextsig_etat reset start cnt bl nb_bloc etat k) *idle*)
      0 1))


(defun nextsig_ram_sel (reset start cnt bl nb_bloc etat k)
  (if (equal  (nextsig_etat reset start cnt bl nb_bloc etat k) *idle*)
      0 1))


(defun nextsig_ram_write (reset start cnt bl nb_bloc etat k)
  (if (or (equal  (nextsig_etat reset start cnt bl nb_bloc etat k)
                  *calcul_abc_one*)
          (equal  (nextsig_etat reset start cnt bl nb_bloc etat k)
                  *resultw*))
      1 0))


(defun nextsig_t (etat t1 l_bloc)
  (if (not (equal etat *idle*))
      (if (and (equal (conv_integer t1) 22)
               (equal etat *calcul_abc_one*))
          (list 0 1 0 0 1 0)
        (if (and (equal etat *init*)
                 (not (equal (conv_integer l_bloc) 0)))
            (list 0 0 0 0 1 0)
          (plus t1 1)))
    (list 0 0 0 0 0 0)))


(defun nextsig_count (etat t1 count)
  (if (equal etat *idle*) (list 0 0 0 0 0 0 0 0)
    (if (equal etat *init*) count
      (if (equal etat *sha_ini_one*) (plus count 1)
        (if (equal etat *calculw_one*) count
          (if (equal etat *calcul_abc_one*)
              (if (equal (conv_integer t1) 22) (plus count 1) count)
            (if (equal etat *resultw*)
                (list 0 0 0 0 0 0 0 0)
              count)))))))


(defun nextsig_cnt (etat t1 cnt)
  (if (equal etat *idle*) 0
```

```
    (if (equal etat *init*)
        (if (>= (conv_integer t1) 0) 1 cnt)
      (if (equal etat *sha_ini_one*) 0
        (if (equal etat *calculw_one*)
            (if (equal (conv_integer t1) 20) 1
                (if (equal (conv_integer t1) 21) 0 cnt))
          (if (equal etat *calcul_abc_one*) cnt
            (if (equal etat *result*)
                (if (equal (conv_integer t1) 19) 1 cnt)
              (if (equal etat *resultw*) 0 cnt)))))))


(defun nextsig_ram_addr
  (reset start etat  base_addr bl nb_bloc k cnt t1 count l_bloc)
  (if (equal (nextsig_etat reset start cnt bl nb_bloc etat k)
             (list 0 1 0))
      (plus (plus base_addr
            (segment 4 8 (nextsig_count etat t1 count)))
            (* (conv_integer
                (nextsig_l_bloc reset start etat k nb_bloc bl))
               16))
    (if (equal (nextsig_etat reset start cnt bl nb_bloc etat k)
               (list 0 1 1))
        (plus (plus base_addr
              (adressage2 (segment 4 8 (nextsig_count etat t1 count))
                          (conv_integer;(nextsig_t etat t1)
                           (nextsig_t etat t1 l_bloc))))
              (* 16 (conv_integer
                     (nextsig_l_bloc reset start etat k nb_bloc bl))))
      (if (equal (nextsig_etat reset start cnt bl nb_bloc etat k)
                 (list 1 0 0))
          (plus (plus  base_addr
                 (segment 4 8 (nextsig_count etat t1 count)))
            (* (conv_integer
                (nextsig_l_bloc reset start etat k nb_bloc bl))
               16))
        base_addr))))


(defun nextsig_wi32 (etat wi32 ram_rdata32 t1)
  (if (equal etat *idle*) wi32
    (if (equal etat *init*) wi32
```

```
      (if (equal etat *sha_ini_one*) ram_rdata32
        (if (equal etat *calculw_one*)
            (if (equal (conv_integer t1) 18) ram_rdata32
              (if (or (equal (conv_integer t1) 19)
                      (equal (conv_integer t1) 20))
                  (b-xor wi32 ram_rdata32)
               (if (equal (conv_integer t1) 21)
                   (shift (b-xor wi32 ram_rdata32)) wi32)))
          wi32)))))


(defun nextsig_a
  (l_bloc a_mem etat t1 ram_rdata32 a b c d e wi32 count)
  (if (equal etat *idle*) a
    (if (equal etat *init*)
        (if (>= (conv_integer t1) 0)
            (if (equal l_bloc
                       (list 0 0 0 0 0 0))
                (hexa-bin (list 6 7 4 5 2 3 0 1)) a) a)
      (if (equal etat *sha_ini_one*)
          (plus (impl-ch b c d) e ram_rdata32
                (append (segment 5 32 a) (segment 0 5 a))
                (hexa-bin (list 5 "a" 8 2 7 9 9 9)))
       (if (equal etat *calculw_one*) a
          (if (equal etat *calcul_abc_one*)
              (if (equal (conv_integer t1) 22)
                  (if (<= (conv_integer count) 19)
                      (plus (ajout2 (impl-ch  b c d) (ajout2 e wi32))
                            (append (segment 5 32 a) (segment 0 5 a))
                            (hexa-bin (list 5 "a" 8 2 7 9 9 9)))
                    (if (and (>= (conv_integer count) 20)
                             (<= (conv_integer count) 39))
                        (plus (b-xor b (b-xor c d))
                              (ajout2 e wi32)
                              (append (segment 5 32 a) (segment 0 5 a))
                              (hexa-bin (list 6 "e" "d" 9 "e" "b" "a" 1)))
                      (if (and (>= (conv_integer count) 40)
                               (<= (conv_integer count) 59))
                          (plus (ajout2 (impl-maj b c d) (ajout2 e wi32))
                                (append (segment 5 32 a) (segment 0 5 a))
                                (hexa-bin
                                 (list 8 "f" 1 "b" "b" "c" "d" "c")))
```

```
                        (plus (b-xor b (b-xor c d))
                              (ajout2 e wi32)
                              (append (segment 5 32 a) (segment 0 5 a))
                              (hexa-bin (list "c" "a" 6 2 "c" 1 "d" 6))))))
                    a)
                (if (equal etat *result*)
                    (if (equal (conv_integer t1) 18)
                        (if (equal l_bloc
                                   (list 0 0 0 0 0 1))
                            (ajout2 a (hexa-bin (list 6 7 4 5 2 3 0 1)))
                         (ajout2 a a_mem))
                      a)
                  a)))))))


(defun nextsig_b (l_bloc b_mem etat a b t1)
  (if (equal etat *idle*) b
    (if (equal etat *init*)
        (if (>= (conv_integer t1) 0)
            (if (equal l_bloc
                       (list 0 0 0 0 0 0))
                (hexa-bin (list "e" "f" "c" "d" "a" "b" 8 9)) b) b)
      (if (equal etat *sha_ini_one*) a
        (if (equal etat *calculw_one*) b
          (if (equal etat *calcul_abc_one*)
              (if (equal (conv_integer t1) 22) a b)
            (if (equal etat *result*)
                (if (equal (conv_integer t1) 18)
                    (if (equal l_bloc
                               (list 0 0 0 0 0 1))
                        (ajout2 b (hexa-bin
                                   (list "e" "f" "c" "d" "a" "b" 8 9)))
                     (ajout2 b b_mem))
                  b) b)))))))


(defun nextsig_c (l_bloc c_mem etat b c t1)
  (if (equal etat *idle*) c
    (if (equal etat *init*)
        (if (>= (conv_integer t1) 0)
            (if (equal l_bloc
                       (list 0 0 0 0 0 0))
```

```
               (hexa-bin (list 9 8 "b" "a" "d" "c" "f" "e")) c) c)
      (if (equal etat *sha_ini_one*)
          (append (segment 30 32 b) (segment 0 30 b))
        (if (equal etat *calculw_one*) c
         (if (equal etat *calcul_abc_one*)
             (if (equal (conv_integer t1) 22)
                 (append (segment 30 32 b) (segment 0 30 b))  c)
           (if (equal etat *result*)
               (if (equal (conv_integer t1) 18)
                   (if (equal l_bloc
                              (list 0 0 0 0 0 1))
                       (ajout2 c (hexa-bin
                                  (list 9 8 "b" "a" "d" "c" "f" "e")))
                    (ajout2 c c_mem))
                  c) c)))))))


(defun nextsig_d (l_bloc d_mem etat c d t1)
  (if (equal etat *idle*) d
    (if (equal etat *init*)
        (if (>= (conv_integer t1) 0)
            (if (equal l_bloc
                       (list 0 0 0 0 0 0))
                (hexa-bin (list 1 0 3 2 5 4 7 6)) d) d)
      (if (equal etat *sha_ini_one*) c
        (if (equal etat *calculw_one*) d
          (if (equal etat *calcul_abc_one*)
              (if (equal (conv_integer t1) 22) c d)
            (if (equal etat *result*)
                (if (equal (conv_integer t1) 18)
                    (if (equal l_bloc
                               (list 0 0 0 0 0 1))
                        (ajout2 d (hexa-bin
                                   (list 1 0 3 2 5 4 7 6)))
                      (ajout2 d d_mem))
                   d) d)))))))


(defun nextsig_e (l_bloc e_mem etat d e t1)
  (if (equal etat *idle*) e
    (if (equal etat *init*)
        (if (>= (conv_integer t1) 0)
```

```
            (if (equal l_bloc
                       (list 0 0 0 0 0 0))
                (hexa-bin (list "c" 3 "d" 2 "e" 1 "f" 0)) e) e)
      (if (equal etat *sha_ini_one*) d
        (if (equal etat *calculw_one*) e
          (if (equal etat *calcul_abc_one*)
              (if (equal (conv_integer t1) 22) d e)
            (if (equal etat *result*)
                (if (equal (conv_integer t1) 18)
                    (if (equal l_bloc
                               (list 0 0 0 0 0 1))
                        (ajout2 e (hexa-bin
                                   (list "c" 3 "d" 2 "e" 1 "f" 0)))
                      (ajout2 e e_mem))
                   e) e)))))))


(defun next_sig_k (etat t1 count)
  (segment 1 8 (nextsig_count etat t1 count)))


(defun nextsig_ram_wdata32 (etat ram_rdata32 t1 wi32 l_bloc)
  (if (equal (conv_integer
              (nextsig_t etat t1 l_bloc)) 22)
      (nextsig_wi32 etat wi32 ram_rdata32 t1)
    (list 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)))


(defun nextsig_aout (etat t1 a aout)
  (if (and (equal etat *result*) (equal (conv_integer t1) 19))
      a aout))


(defun nextsig_bout (etat t1 b bout)
  (if (and (equal etat *result*) (equal (conv_integer t1) 19))
      b bout))


(defun nextsig_cout (etat t1 c cout)
  (if (and (equal etat *result*) (equal (conv_integer t1) 19))
      c cout))
```

```
(defun nextsig_dout (etat t1 d dout)
  (if (and (equal etat *result*) (equal (conv_integer t1) 19))
      d dout))


(defun nextsig_eout (etat t1 e eout)
  (if (and (equal etat *result*) (equal (conv_integer t1) 19))
      e eout))


(defun nextsig_a_mem (etat t1 a a_mem)
  (if (and (equal etat *result*) (equal (conv_integer t1) 19))
      a a_mem))


(defun nextsig_b_mem (etat t1 b b_mem)
  (if (and (equal etat *result*) (equal (conv_integer t1) 19))
      b b_mem))


(defun nextsig_c_mem (etat t1 c c_mem)
  (if (and (equal etat *result*) (equal (conv_integer t1) 19))
      c c_mem))


(defun nextsig_d_mem (etat t1 d d_mem)
  (if (and (equal etat *result*) (equal (conv_integer t1) 19))
      d d_mem))


(defun nextsig_e_mem (etat t1 e e_mem)
  (if (and (equal etat *result*) (equal (conv_integer t1) 19))
      e e_mem))
(set-ignore-ok t)


(defun sim-step (input mem)
  (let ((reset        (nth 1 input))    ; Input,     bit
        (reset_done   (nth 2 input))    ; input,     bit
        (start        (nth 3 input))    ; input,     bit
        (base_addr    (nth 4 input))    ; input,     12-bit
        (nb_bloc      (nth 5 input))    ; input,     6-bit
        (ram_rdata32  (nth 0 input))    ; input,     32-bit

        (a            (nth 0 mem))    ; Internal,  32-bit
```

```
    (b            (nth 1 mem))    ; Internal,  32-bit vector
    (c            (nth 2 mem))    ; Internal,  32-bit
    (d            (nth 3 mem))    ; Internal,  32-bit
    (e            (nth 4 mem))    ; Internal,  32-bit
    (aout         (nth 5 mem))    ; Internal,  32-bit
    (bout         (nth 6 mem))    ; Internal,  32-bit vector
    (cout         (nth 7 mem))    ; Internal,  32-bit
    (dout         (nth 8 mem))    ; Internal,  32-bit
    (eout         (nth 9 mem))    ; Internal,  32-bit
    (a_mem        (nth 10 mem))   ; Internal,  32-bit
    (b_mem        (nth 11 mem))   ; Internal,  32-bit vector
    (c_mem        (nth 12 mem))   ; Internal,  32-bit
    (d_mem        (nth 13 mem))   ; Internal,  32-bit
    (e_mem        (nth 14 mem))   ; Internal,  32-bit
    (wi32         (nth 15 mem))   ; Internal,  32-bit
    (t1           (nth 16 mem))   ; Internal,  6-bit
    (count        (nth 17 mem))   ; Internal,  8-bit
    (bl           (nth 18 mem))   ; Internal,  6-bit
    (k            (nth 19 mem))   ; Internal,  7-bit
    (etat         (nth 20 mem))   ; Internal,  3-bit vector
    (cnt          (nth 21 mem))   ; Internal,  bit
    (l_bloc       (nth 22 mem))   ; Internal,  6-bit

    (ram_addr     (nth 23 mem))  ; Outputs,   12-bit
    (ram_wdata32  (nth 24 mem))  ; Outputs,   32-bit
    (ram_sel      (nth 25 mem))  ; Outputs,   bit
    (ram_write    (nth 26 mem))  ; Outputs,   bit
    (busy         (nth 27 mem))  ; Outputs,   bit
    (done         (nth 28 mem))  ; Outputs,   bit)

(list
 (nextsig_a l_bloc a_mem etat t1 ram_rdata32 a b c d e wi32 count)
 (nextsig_b l_bloc b_mem etat a b t1)
 (nextsig_c l_bloc c_mem etat b c t1)
 (nextsig_d l_bloc d_mem etat c d t1)
 (nextsig_e l_bloc e_mem etat d e t1)
 (nextsig_aout etat t1 a aout)
 (nextsig_bout etat t1 b bout)
 (nextsig_cout etat t1 c cout)
 (nextsig_dout etat t1 d dout)
 (nextsig_eout etat t1 e eout)
 (nextsig_a_mem etat t1 a a_mem)
```

```
     (nextsig_b_mem etat t1 b b_mem)
     (nextsig_c_mem etat t1 c c_mem)
     (nextsig_d_mem etat t1 d d_mem)
     (nextsig_e_mem etat t1 e e_mem)
     (nextsig_wi32 etat wi32 ram_rdata32 t1)
     (nextsig_t etat t1 l_bloc)
     (nextsig_count etat t1 count)
     (nextsig_bl reset start etat k nb_bloc bl)
     (next_sig_k etat t1 count)
     (nextsig_etat reset start cnt bl nb_bloc etat k)
     (nextsig_cnt etat t1 cnt)
     (nextsig_l_bloc reset start etat k nb_bloc bl)
     (nextsig_ram_addr
       reset start etat base_addr bl nb_bloc k cnt t1 count l_bloc)
     (nextsig_ram_wdata32 etat ram_rdata32 t1 wi32 l_bloc)
     (nextsig_ram_sel reset start cnt bl nb_bloc etat k)
     (nextsig_ram_write reset start cnt bl nb_bloc etat k)
     (nextsig_busy reset start cnt bl nb_bloc etat k)
     (nextsig_done reset reset_done start etat cnt bl done))))

(defconst *a* 0)
(defconst *b* 1)
(defconst *c* 2)
(defconst *d* 3)
(defconst *e* 4)
(defconst *aout* 5)
(defconst *bout* 6)
(defconst *cout* 7)
(defconst *dout* 8)
(defconst *eout* 9)
(defconst *a_mem* 10)
(defconst *b_mem* 11)
(defconst *c_mem* 12)
(defconst *d_mem* 13)
(defconst *e_mem* 14)
(defconst *wi32* 15)
(defconst *t1* 16)
(defconst *count* 17)
(defconst *bl* 18)
(defconst *k* 19)
(defconst *etat* 20)
(defconst *cnt* 21)
```

```
(defconst *l_bloc* 22)
(defconst *ram_addr* 23)
(defconst *ram_wdata32* 24)
(defconst *ram_sel* 25)
(defconst *ram_write* 26)
(defconst *busy* 27)
(defconst *done* 28)

(defconst *reset* 0)
(defconst *reset_done* 1)
(defconst *start* 2)
(defconst *base_addr* 3)
(defconst *nb_bloc* 4)
(defconst *ram_rdata32* 5)


(defun read-ram (mem ram)
  (if (equal (nth *ram_sel* mem) 1)
      (binding-equal (nth *ram_addr* mem) ram)
    nil))


(defun write-ram (mem ram)
  (if (and (equal (nth *ram_sel* mem) 1)
           (equal (nth *ram_write* mem) 1))
      (bind-equal (nth *ram_addr* mem)
                  (nth *ram_wdata32* mem) ram)
    ram))


(defun sha_vhdl (l-input st)
  (if (atom l-input)
      st
    (let* ((memory (car st))
           (ram (cdr st))
           (new-mem
            (sim-step (cons (read-ram memory ram)
                            (car l-input))
                      memory)))
      (sha_vhdl (cdr l-input)
                (cons new-mem
                      (write-ram new-mem ram))))))
```

Bibliographie

[1] J.-R. Abrial. *The B Book - Assigning programs to meanings.* Cambridge University Press, 1996.

[2] J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In Didier Bert, editor, *B'98 : Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. Springer, 1998.

[3] A. Aljer, P. Devienne, S. Tison, J.-L. Boulanger, and G. Mariano. B-HDL : Circuit Design in B. In *In ACSD 2003, International Conference on Application of Concurrency to System Design*, pages 241–242, 2003.

[4] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2002.

[5] G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. Bounded Model Checking for Timed Systems. In *FORTE '02 : Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, pages 243–259, London, UK, 2002. Springer-Verlag.

[6] T. Ball and S. K. Rajamani. The SLAM project : debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[7] M. Ben-Ari, Z. Manna, and A. Pnueli. The Temporal Logic of Branching Time. In *POPL*, pages 164–176, 1981.

[8] B. Bentley. Validating the Intel Pentium 4 Microprocessor. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 244–248. ACM, 2001.

[9] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art : The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[10] Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors. *Theorem Proving in Higher Order Logics, 12th International Conference,*

TPHOLs'99, Nice, France, September, 1999, Proceedings, volume 1690 of *Lecture Notes in Computer Science*. Springer, 1999.

[11] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. *Lecture Notes in Computer Science*, 1579 :193–207, 1999.

[12] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *DAC '99 : Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 317–320, New York, NY, USA, 1999. ACM Press.

[13] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58(3), 2003.

[14] C. Blank. Formal methods and their application. In *Workshop on Industrial Application of Formal Methods, Munich*, 2005.

[15] D. Borrione and W. J. Paul, editors. *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, volume 3725 of *Lecture Notes in Computer Science*. Springer, 2005.

[16] D. Borrione, R. Piloty, D. Hill, K. J. Lieberherr, and P. Moorby. Three Decades of HDLs : Part II, Conlan Through Verilog. *IEEE Des. Test*, 9(3) :54–63, 1992.

[17] D. Borrione and A. M. Salem. Denotational Semantics of a Synchronous VHDL Subset. *Formal Methods in System Design*, 7(1/2) :53–71, 1995.

[18] R. J. Boulton, A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with Embedding Hardware Description Languages in HOL. In *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design : Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992.

[19] B. Brock, M. Kaufmann, and J S. Moore. ACL2 Theorems About Commercial Microprocessors. In *FMCAD '96 : Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 275–293, London, UK, 1996. Springer-Verlag.

[20] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[21] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2002.

[22] R. E. Bryant and C.-J. H. Seger. Formal Verification of Digital Circuits Using Symbolic Ternary System Models. In Clarke and Kurshan [34], pages 33–43.

[23] D. Cachera and D. Pichardie. Embedding of Systems of Affine Recurrence Equations in Coq. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, volume 2758 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2003.

[24] D. Cachera, P. Quinton, S. Rajopadhye, and T. Risset. Proving Properties of Multi-dimensional Recurrences with Application to Regular Parallel Algorithms. In *FMPP-TA'01*, San Francisco, CA, April 2001.

[25] P. Camurati and H. Eveking, editors. *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95, Frankfurt/Main, Germany, October 2-4, 1995, Proceedings*, volume 987 of *Lecture Notes in Computer Science*. Springer, 1995.

[26] W. C. Carter, W. H. Joyner, and D. Brand. Symbolic simulation for correct machine design. In *DAC '79 : Proceedings of the 16th Conference on Design automation*, pages 280–286, Piscataway, NJ, USA, 1979. IEEE Press.

[27] Y. Chu, D. L. Dietmeyer, J. R. Duley, F. J. Hill, M. R. Barbacci, C. W. Rose, G. Ordy, B. Johnson, and M. Roberts. Three Decades of HDLs : Part I, CDL Through TI-HDL. *IEEE Des. Test*, 9(2) :69–81, 1992.

[28] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1) :7–34, 2001.

[29] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In Kozen [78], pages 52–71.

[30] 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 *POPL*, pages 117–126, 1983.

[31] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[32] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5) :1512–1542, September 1994.

[33] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[34] E.M. Clarke and R. P. Kurshan, editors. *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, volume 531 of *Lecture Notes in Computer Science*. Springer, 1991.

[35] O. Coudert, J. C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams. In Clarke and Kurshan [34], pages 23–32.

[36] S. Coupet-Grimal and L. Jakubiec. Hardware Verification Using Co-induction in COQ. In Bertot et al. [10], pages 91–108.

[37] D. Deharbe. *Vérification formelle de propriétés temporelles : Étude et Application au Langage VHDL*. PhD thesis, Université Joseph Fourier, 1996.

[38] D. Deharbe. A tutorial introduction to symbolic model checking. In *Logic for concurrency and synchronisation*, pages 215–237, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[39] Al Dewey. VHSIC hardware description (VHDL) development program. In *DAC '83 : Proceedings of the 20th conference on Design automation*, pages 625–628, Piscataway, NJ, USA, 1983. IEEE Press.

[40] M. Dezani-Ciancaglini and U. Montanari, editors. *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *Lecture Notes in Computer Science*. Springer, 1982.

[41] H. Dobbertin. Cryptanalysis of MD4. In Dieter Gollmann, editor, *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 1996.

[42] E. Dumitrescu. *Construction de modèles réduits et vérification symbolique de circuits industriels décrits au niveau RTL*. PhD thesis, Université Joseph Fourier, 2003.

[43] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, pages 995–1072. Elsevier, 1990.

[44] E. Encrenaz. *Une méthode de vérification de propriétés de programmes VHDL basée sur des réseaux de Petri*. PhD thesis, Université Paris VI, 1995.

[45] P. Frison, F. Charot, E. Gautrin, D. Lavenier, P. Quinton, F. Raimbault, and C. Wagner. From Equations to Hardware. Towards the Systematic Mapping of Algorithms onto Parallel Architectures. *IJPRAI*, 8(2) :417–438, 1994.

[46] E. Gat. Point of view : Lisp as an alternative to Java. *Intelligence*, 11(4) :21–24, 2000.

[47] P. Georgelin. *Vérification formelle de systémes digitaux synchrones, basée sur la simulation symbolique*. PhD thesis, Université Joseph Fourier, 2001.

[48] A. Gill. *Introduction to the theory of finite state machines*. Electronic Science Series. McGraw Hill, 1962.

[49] K. G. W. Goossens. Reasoning about VHDL using operational and observational semantics. In Camurati and Eveking [25], pages 311–327.

[50] D. Greve. Symbolic simulation of the JEM1 microprocessor. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522, pages 321–333, Palo Alto, CA, 1998. Springer-Verlag.

[51] A.-C. Guillou, P. Quinton, and T. Risset. Hardware Synthesis for Multi-Dimensional Time. In *14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2003), 24-26 June 2003, The Hague, The Netherlands*, pages 40–50. IEEE Computer Society, 2003.

[52] N. Halbwachs and D. Peled, editors. *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*. Springer, 1999.

[53] T. A. Henzinger, O. Kupferman, and S. Qadeer. From *re*-historic to *ost*-modern Symbolic Model Checking. In Hu and Vardi [54], pages 195–206.

[54] A. J. Hu and M. Y. Vardi, editors. *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.

[55] D. Hutter, W. Stephan, P. Traverso, and M. Ullman, editors. *PVS : An Experience Report*, volume 1641 of *Lecture Notes in Computer Science*, Boppard, Germany, oct 1998. Springer-Verlag.

[56] IEEE. 1076-1987 IEEE Standard VHDL Reference Manual, 1988.

[57] IEEE. 1076-1993 IEEE Standard VHDL Language Reference Manual, 1994.

[58] IEEE. 1364-1995 IEEE Standard Verilog Hardware Description Language Based on the Verilog Hardware Description Language, 1996.

[59] IEEE. 1076-2000 IEEE Standard VHDL Language Reference Manual, 2000.

[60] IEEE. 1364-2001 IEEE Standard Verilog Hardware Description Language, 2001.

[61] IEEE. 1076.6-2004 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, 2004.

[62] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessors Control. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 68–80, Standford, California, USA, 1994. Springer-Verlag.

[63] A. Jerraya. *Conception de haut niveau des systèmes monopuces*. HERMES Science Publications, 2005.

[64] J.Kim, A. Biryukov, B. Preneel, and S. Lee. On the Security of Encryption Modes of MD4, MD5 and HAVAL. In S. Qing, W. Mao, J. Lopez, and G. Wang, editors, *Information and Communications Security, 7th International Conference, ICICS 2005, Beijing, China, December 10-13, 2005, Proceedings*, volume 3783 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 2005.

[65] W. A. Hunt Jr. and S. D. Johnson, editors. *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*. Springer, 2000.

[66] W. A. Hunt Jr. and E. Reeber. Formalization of the DE2 Language. In Borrione and Paul [15], pages 20–34.

[67] W. A. Hunt Jr. and J. Sawada. Verifying the FM9801 Microarchitecture. *IEEE Micro*, 19(3) :47–55, 1999.

[68] 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.

[69] D. Kahn. *The Codebreakers : The Story of Secret Writing*. Scribner, 1996.

[70] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, 1987. Springer-Verlag.

[71] R. M. Karp, R. E. Miller, and S. Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the ACM (JACM)*, 14(3), 1967.

[72] M. Kaufmann, P. Manolios, and J S. Moore. *ACL2 Computer Aided Reasoning : An Approach*. Kluwer Academic Press, 2000.

[73] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning : ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.

[74] M. Kaufmann and J S. Moore. A Precise Description of the ACL2 Logic. In *Research Report*. Dept. of Computer Sciences, University of Texas at Austin, 1997.

[75] M. Kaufmann and J S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Trans. Softw. Eng.*, 23(4) :203–213, 1997.

[76] M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *J. Autom. Reasoning*, 26(2) :161–203, 2001.

[77] C. D. Kloos. *Formal Semantics for VHDL*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[78] D. Kozen, editor. *Logic of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1982.

[79] A. Kuehlmann and F. Krohm. Equivalence Checking Using Cuts and Heaps. In *DAC*, pages 263–268, 1997.

[80] S. K. Lahiri and S. A. Seshia. The UCLID Decision Procedure. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478. Springer, 2004.

[81] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Software Eng.*, 3(2) :125–143, 1977.

[82] L. Lamport. "sometime" is sometimes "not never" - on the temporal logic of programs. In *POPL*, pages 174–185, 1980.

[83] S. Y. Liao, S. W. K. Tjiang, and R. K. Gupta. An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment. In *DAC*, pages 70–75, 1997.

[84] P. Manolios and J. Strother Moore. Partial Functions in ACL2. *Journal of Automated Reasoning*, 31(2) :107–127, 2003.

[85] P. Manolios, K. S. Namjoshi, and R. Summers. Linking Theorem Proving and Model-Checking with Well-Founded Bisimulation. In Halbwachs and Peled [52], pages 369–379.

[86] C. Mauras. *ALpha : un langage équationnel pour la conception et la programmation d'architectures systoliques*. PhD thesis, Université Rennes I France, December 1989.

[87] J. L. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM*, 3(4) :184–195, 1960.

[88] K. L. McMillan. *Symbolic model checking : an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[89] T.P. Melham. Abstraction Mechanisms for hardware verification. In G. Birtwistle ans P. Subrahmanyam, editor, *VLSI Specification, Verification and Synthesis*, pages 267–291. Kluwer Academic Publishers, 1988.

[90] J. Mermet, editor. *UML-B Specification for Proven Embedded Systems Design*. Kluwer Academic Publishers, 2004.

[91] J S. Moore. Introduction to the OBDD Algorithm for the ATP Community. *Journal of Automated Reasoning*, 12(1) :33–46, 1994.

[92] J S. Moore. Symbolic Simulation : An ACL2 Approach. In *FMCAD '98 : Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, pages 334–350, London, UK, 1998. Springer-Verlag.

[93] J S. Moore. Rewriting for Symbolic Execution of State Machine Models. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 411–422. Springer, 2001.

[94] K. Morin-Allory and D. Borrione. Proven correct monitors from PSL specifications. In *DATE*. IEEE, 2006.

[95] NSA. FIPS 180 : Secure Hash Standard, 1991.

[96] NSA. FIPS 180-1 : Secure Hash Algorithm, 1993.

[97] NSA. FIPS 180-2 : Secure Hash Algorithm, 2002.

[98] S. Olcoz and J. M. Colom. A Colored Petri Net Model of VHDL. *Formal Methods in System Design*, 7(1/2) :101–123, 1995.

[99] S. Owre, J. M. Rushby, , and N. Shankar. PVS : A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[100] S. Owre and N. Shankar. Writing PVS Proof Strategies. In M. Archer, B. Di Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number CP-2003-212448 in NASA Conference Publication, pages 1–15, Hampton, VA, September 2003. NASA Langley Research Center.

[101] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61 :17–139, 2004.

[102] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Dezani-Ciancaglini and Montanari [40], pages 337–351.

[103] F. Quilleré and S. V. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst.*, 22(5) :773–815, 2000.

[104] F. Quilleré, S. V. Rajopadhye, and D. Wilde. Generation of Efficient Nested Loops from Polyhedra. *International Journal of Parallel Programming*, 28(5) :469–498, 2000.

[105] P. Quinton. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. In *ISCA*, pages 208–214, 1984.

[106] S. Ray. Attaching Efficient Executability to Partial Functions in ACL2. In M. Kaufmann and J S. Moore, editors, *5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Austin, TX, November 2004.

[107] S. Ray and W. A. Hunt Jr. Deductive Verification of Pipelined Machines Using First-Order Quantification. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2004.

[108] S. Read and M. Edwards. A Formal Semantics of VHDL in Booyer-Moore Logic. In CA San Diego, editor, *2nd International Conference on Concurrent Engineering and EDA*. SCSI, 1994.

[109] I. S. Reed. Symbolic synthesis of digital computers. In *ACM '52 : Proceedings of the 1952 ACM national meeting (Toronto)*, pages 90–94, New York, NY, USA, 1952. ACM Press.

[110] R. Reetz and T. Kropf. A Flowgraph Semantics of VHDL : Toward a VHDL Verification Workbench in HOL. *Formal Methods in System Design*, 7(1/2) :73–99, 1995.

[111] R. Rivest. The MD5 Message-Digest Algorithm. In *RFC 1321*. MIT and RSA Data Security, Inc, 1992.

[112] R. L. Rivest. The MD4 Message Digest Algorithm. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1990.

[113] D. M. Russinoff. A Formalization of a Subset of VHDL in the Boyer-Moore Logic. *Formal Methods in System Design*, 7(1/2) :7–25, 1995.

[114] D. M. Russinoff. A Case Study in Fomal Verification of Register-Transfer Logic with ACL2 : The Floating Point Adder of the AMD Athlon^{TM} Processor. In Jr. and Johnson [65], pages 3–36.

[115] G. Al Sammane. *Simulation symbolique des circuits décrits au niveau algorithmique*. PhD thesis, Université Joseph Fourier, July 18, 2005.

[116] G. Al Sammane, D. Borrione, P. Ostier, J. Schmaltz, and D. Toma. Combining ACL2 and Mathematica for the Symbolic Simulation of Digital Systems. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, July 2003.

[117] G. Al Sammane, J. Schmaltz, D. Toma, P. Ostier, and D. Borrione. TheoSim : combining symbolic simulation and theorem proving for hardware verification. In New York ACM, editor, *SBCCI : 17th Symposium on Integrated Circuits and Systems Design IEEE*, 2004.

[118] G. Al Sammane, D. Toma, J. Schmaltz, P. Ostier, and D. Borrione. Constrained Symbolic Simulation with Mathematica and ACL2. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, volume 2860 of *Lecture Notes in Computer Science*, pages 150–157. Springer, 2003.

[119] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *STTT*, 5(2-3) :185–204, 2004.

[120] L. Séméria, K. Sato, and G. De Micheli. Resolution of Dynamic Memory Allocation and Pointers for the Behavioral Synthesis from C. In *2000 Design, Automation and Test in Europe (DATE 2000), 27-30 March 2000, Paris, France*, pages 312–319. IEEE Computer Society, 2000.

[121] J. R. Shoenfield. *Mathematical logic*. Addison Wesley, 1967.

[122] R. D. Tennent. *Semantics of Programming Languages*. Prentice Hall, 1991.

[123] K. Thirunarayan and R. L. Ewing. Structural Operational Semantics for a Portable Subset of Behavioral VHDL-93. *Formal Methods in System Design*, 18(1) :69–88, 2001.

[124] D. Toma and D. Borrione. SHA Formalization. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, July 2003.

[125] D. Toma and D. Borrione. Verification of a Cryptographic Circuit : SHA-1 using ACL2. In *Fifth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2004)*, November 2004.

[126] D. Toma and D. Borrione. Formal Verification of a SHA-1 Circuit Core Using ACL2. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2005.

[127] D. Toma, D. Borrione, and Ghiath Al Sammane. Combining Several Paradigms for Circuit Validation and Verification. In G. Barthe, L. Burdy, M. Huisman, J.-L.

Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, volume 3362 of *Lecture Notes in Computer Science*, pages 229–249. Springer, 2005.

[128] D. Toma, A. Perez, D. Borrione, and E. Bergeret. Design of a proven correct SHA circuit. In IEEE, editor, *International Conference on Electrical, Electronic and Computer Engineering, ICEEC-04, Cairo, Egypt*, 2004.

[129] J. van Tassel. A Formalisation of the VHDL Simulation Cycle. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications, Proceedings of the IFIP TC10/WG10.2 Workshop HOL'92, Leuven, Belgium, 21-24 September 1992*, volume A-20 of *IFIP Transactions*, pages 359–374. North-Holland/Elsevier, 1993.

[130] J. van Tassel and D. Hemmendinger. Toward Formal Verification of VHDL Specifications. In *In International Workshop on Applied Formal Methods for Correct VLSI Design*, IFIP WG 10.2/WG 10.5, pages 409–418. North-Holland, 1990.

[131] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings, Symposium on Logic in Computer Science, 16-18 June 1986, Cambridge, Massachusetts, USA*, pages 332–344. IEEE Computer Society, 1986.

[132] M. N. Velev. Using Automatic Case Splits and Efficient CNF Translation to Guide a SAT-solver when Formally Verifying Out-Of-Order Processors. In *AI&M 1-2004, Eighth International Symposium on Artificial Intelligence and Mathematics, January 4-6, 2004, Fort Lauderdale, Florida, USA*, 2004.

[133] M. N. Velev and R. E. Bryant. EVC : A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality, and Conservative Transformations. In *CAV '01 : Proceedings of the 13th International Conference on Computer Aided Verification*, pages 235–240, London, UK, 2001. Springer-Verlag.

[134] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In V. Shoup, editor, *Advances in Cryptology - CRYPTO 2005 : 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[135] X. Wang, H. Yu, and Y. L. Yin. Efficient Collision Search Attacks on SHA-0. In V. Shoup, editor, *Advances in Cryptology - CRYPTO 2005 : 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[136] D. Wilde and S. V. Rajopadhye. The naive execution of affine recurrence equations. In *Proceedings of the IEEE International Conference on Application Specific Array Processors*. IEEE Computer Society Washington, DC, USA, 1995.

[137] D. Wilde and S. V. Rajopadhye. Memory Reuse Analysis in the Polyhedral Model. In *Euro-Par, Vol. I*, pages 389–397, 1996.

[138] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. Cost-Efficient Block Verification for a UMTS Up-Link Chip-Rate Coprocessor. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, pages 162–167. IEEE Computer Society, 2004.

[139] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

[140] X.Wang and H. Yu. How to Break MD5 and Other Hash Functions. In R. Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[141] Y. Zimmermann and D. Toma. Component Reuse in B Using ACL2. In H. Treharne, S. King, M. C. Henson, and S. Schneider, editors, *ZB 2005 : Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, volume 3455 of *Lecture Notes in Computer Science*, pages 279–298. Springer, 2005.

Résumé

A cause de la complexité croissante des systèmes sur puce (SoC), la vérification devient un aspect très important : 70 - 80% du coût de conception est alloué à cette tâche. Plus de 60% des projets de développement d'ASIC doivent être repris à cause des erreurs fonctionnelles, environ 50% des erreurs de conception étant situées au niveau du module. Dans le monde industriel, la vérification est souvent synonyme de simulation - une méthode de vérification naturelle pour les concepteurs, mais qui ne garantit pas l'absence d'erreurs. Une alternative est fournie par la vérification formelle qui prouve mathématiquement qu'un circuit satisfait une spécification. Dans cette thèse, on s'intéresse aux méthodes déductives basées sur la démonstration de théorèmes. La démonstration de théorèmes permet de vérifier formellement des descriptions matérielles de haut niveau et des systèmes réguliers ou très complexes, car la taille de données n'a plus d'importance. Par contre la modélisation de la description matérielle se fait directement en logique, ce qui rend l'accès difficile pour les concepteurs. Notre travail a pour but de faciliter l'introduction des outils de démonstration de théorèmes dans le flot de conception. Nous proposons une méthode automatique de traduction d'un circuit VHDL vers un modèle sémantique basé sur des équations récurrentes par rapport au temps qui peut être l'entrée de tout outil de démonstration de théorèmes et nous définissons une approche de vérification adaptée au modèle. Afin de valider notre proposition, nous avons choisi le démonstrateur ACL2 pour vérifier une bibliothèque de circuits de cryptographie.

Mots clés : méthodes formelles, démonstration de théorèmes, système sur puce, composants cryptographiques

Abstract

Due to the growing complexity of SoC, the verification became a very important aspect : 70-80% of the design cost is allocated to this task. More than 60% of the ASIC development projects have to be remade because of functional errors - 50% of the functional errors are at the module level. In the industry, verification is synonym with simulation - a natural verification method for the designers, but it does not guarantee the absence of the errors. An alternative is formal verification, which proves mathematically that a circuit satisfies a specification. In this thesis we are interested in deductive methods based on theorem proving. Theorem proving is generally used for the formal verification of high level or complex designs, as the size of data is not important. The inconvenient is that the model of the design is described directly in the logic of the tool, which makes the access to the technique very difficult for the designers. The goal of our work is to facilitate the introduction of such tools in the design flow. We propose an automatic method to translate a VHDL design to a semantic model based on recurrent equations on time, which can be the input to any theorem proving tool and we define a verification approach adapted to the model. To validate our proposal, we chose ACL2 as tool to verify a cryptographic library.

Keywords : formal methods, theorem proving, system on chip, cryptographic IPs

ISBN : 2-84813-087-3