# Vérification formelle de systèmes digitaux synchrones, basée sur la simulation symbolique

P. Georgelin

Thèse

Présentée par

**Philippe Georgelin**

pour obtenir le grade de

Docteur de l'Université Joseph Fourier- Grenoble 1

(arrêté ministériel du 30 Mars 1992)

(Spécialité **Informatique)**

# Vérification formelle de systèmes digitaux synchrones, basée sur la simulation symbolique

**Date de soutenance: 18 Octobre 2001**

**Composition du Jury:**

| | | |
|---|---|---|
| Mesdames | | |
| | Dominique Borrione, Prof., Université Joseph Fourier | Directeur de thèse |
| | Laurence Pierre, M.C.HDR, Université de Provence | |
| Messieurs | | |
| | Nicolas Halbwachs, Direct. de Recherche CNRS | Président |
| | Przemyslaw Bakowski, Prof., Université de Nantes | Rapporteur |
| | Hans Eveking, Prof., Université de Darmstadt | Rapporteur |
| | Henri Michel, STMicroelectronics, Central R&D Crolles | |

Thèse préparée au laboratoire **Techniques de l'Informatique et de la Microélectronique pour l'Architecture d'ordinateurs (TIMA)**

# Remerciements

Je voudrais remercier Madame Dominique Borrione, qui a assuré la direction de ma thèse, pour m'avoir accueilli au sein de l'équipe VDS, pour son suivi régulier et ses conseils judicieux.

Un grand merci à mes rapporteurs: Monsieur Pete Bakowsky, de l'Université de Nantes et Monsieur Hans Eveking, de l'Université Technologique de Darmstadt, pour avoir accepté de rapporter ce travail malgré les délais courts en période de vacances.

Je remercie Monsieur Nicolas Halbwachs pour l'honneur qu'il me fait en présidant le Jury.

Un très grand merci à Laurence Pierre, de l'Université de Provence, qui, à partir du premier stage d'été en 1996, n'a cessé de m'apporter aide, soutien et connaissances. Merci également à Henri Michel, de STMicroelectronics, de s'être intéressé de près à mon travail.

Je dois beaucoup à Pierre Ostier, pour ses conseils, son soutien et ses nombreuses "blagues".

J'exprime toutes mes amitiés aux autres membres de l'équipe VDS qui m'ont accompagné durant ces trois années de thèse : Claude Le Faou, Emil Dumitrescu, Julia Dushina, Adam Morawiec, Menouer Boubekeur, Ghiath Al Sammane, Christophe Boursin et à tous ceux que je ne peux citer du Laboratoire TIMA.

Je tiens à exprimer ma gratitude à tous ceux qui m'ont aidé et soutenu et à mes amis : Charles, Christelle, Stéphane, Alain, Régine, Vincente, Patrick, Brigitte et.... troupy !

Une mention spéciale pour Vanderlei Moraes Rodrigues qui m'a efficacement encadré et guidé pendant ma deuxième année de thèse.

Pour finir, ma reconnaissance va à mes parents qui m'ont encouragé et soutenu tout au long de mes études.

## Résumé

La plupart des outils de vérification formelle comme les "Model-checkers" sont restrictifs car ils ne peuvent travailler avec des niveaux plus haut que le "RTL", et ils sont également limités sur le nombre total d'états. Les démonstrateurs de théorèmes ne souffrent pas de ces restrictions, mais ne sont pas automatiques et requièrent des méthodes pour faciliter leur utilisation systématique.

Cette thèse aborde la vérification formelle de descriptions VHDL au moyen du démonstrateur ACL2. Nous proposons un environnement combinant simulation symbolique et démonstrateur de théorèmes pour l'analyse formelle de descriptions de haut niveau d'abstraction.

Plus précisément, notre approche consiste à développer des méthodes
- pour formaliser un sous-ensemble de VHDL,
- pour "diriger" le démonstrateur pour effectuer de la simulation symbolique
- pour utiliser ces résultats pour les preuves.

Un outil a été développé combinant des traducteurs (VHDL vers ACL2), des moteurs de simulation symbolique et de preuves, et une interface utilisateur. Les définitions et les théorèmes sont générés automatiquement. Un même modèle généré est ainsi utilisé pour toutes les tâches.

Nous aspirons à fournir au concepteur une méthodologie pour insérer la vérification formelle le plus tôt possible dans le cycle de conception. Le démonstrateur est utilisé pour des manipulations symboliques et pour prouver qu'ils sont équivalents à une fonction spécifiée.

Le résultat de cette thèse est de rendre la technique de démonstration de théorèmes acceptable dans une équipe de concepteur du point de vue de la facilité d'utilisation, et de diminuer le temps de vérification.


**Mot Clés:** Vérification formelle, démonstration de théorèmes, Acl2, simulation symbolique.


## Abstract

To satisfy market requirements, formal verification tools must allow designers to verify complex descriptions and reason about large or infinite sets of values. One should be able to concentrate on the correctness of algorithms and the essential mathematical properties of the blocks being designed.

Most modern verification tools such as Model Checkers are restrictive because they can't deal with abstraction levels higher than Register Transfer Level, or similar Finite-State Machine models and are also limited on the total number of states. Theorem provers do not suffer from these restrictions, but they are not fully automated, and require methods to ease their systematic use in the standard design flow.

This thesis addresses the formal verification of VHDL descriptions with the ACL2 theorem prover. We propose an environment combining symbolic simulation and theorem proving for the formal analysis of high level VHDL designs.

Our approach consists in developping methods
- to formalize a synthesis subset of VHDL,
- to "direct" the theorem prover to perform symbolic simulation
- to use symbolic simulation results for proofs.

A tool was developped combining translators from VHDL to ACL2, symbolic simulation and proof engines in a user interface. The definitions and theorems that formalize the VHDL input are generated automatically, and the resulting model is executable. This same model is used for symbolic simulation and proof.

By combining symbolic simulation and theorem proving, we aim at providing the verification engineer with a methodology to efficiently insert formal verification in the very early specification stages of a design. The theorem prover can be used to perform symbolic manipulations on the result expressions, and prove that they are equivalent to a specified function.

The result of this thesis is to make theorem proving techniques more acceptable to a design team in terms of ease of use, and to notably decrease verification time in a design process.

# Table des matières

# Chapitre 1

# Introduction

## 1.1 Problème étudié dans la thèse

Au cours des dernières décennies, la complexité des circuits intégrés n'a cessé de croître. Les progrès technologiques, notamment en terme de finesse de gravure, ont permis de diminuer considérablement les tailles des circuits intégrés. Cette miniaturisation entraîne d'énormes avantages, tant au niveau de leur consommation (utilisation dans les ordinateurs portables, téléphones cellulaires, ...), de leur puissance (on complexifie les circuits en leur ajoutant plus de fonctions) et de leur rapidité (les fréquences d'horloges peuvent être augmentées grâce à cette miniaturisation).
Lors de la conception des circuits intégrés, la phase de *vérification* est l'étape la plus longue. Elle représente 40 à 70 % du temps de conception. Une prise de conscience de l'importance de cette phase a émergé lors de la médiatisation de l'erreur de conception du Pentium d'Intel en 1994 [29, 28][1]. La perte a été estimée à 400 millions de dollars. Ce cas n'est pas isolé, de nombreux exemples d'erreurs de conception se trouvent actuellement (65 "bugs" répertoriés dans le Pentium III [30], 6 dans l'AMD Athlon [3], erreurs de conception dans les revisions 2.2 à 2.6 du PowerPC 7400 (G4) [46].)

Actuellement, les circuits généralistes (comme les microprocesseurs), mais aussi les circuits spécialisés (télécommunications, traitement de l'image, ...) deviennent de plus en plus élaborées rendant le processus de conception plus complexe et favorisant ainsi l'accroissement du nombre d'erreurs de conception possibles, à tous les niveaux du processus, depuis l'analyse du cahier des charges jusqu'à la fabrication du circuit lui-même. Il est donc de plus en plus crucial de disposer de méthodes permettant de s'assurer de la validité d'un système avant d'en arriver à cette phase ultime, très coûteuse.
*Garantir, de façon efficace et sûre, la fiabilité d'un circuit , le plus tôt possible lors de sa mise au point, représente actuellement un enjeu économique important.*
Ce mémoire présente une méthode de vérification de circuits basée sur la technique de démonstration de théorèmes. Les applications de cette méthode envisagées sont principalement les

1. L'opérateur de division FDIV fournit des résultats incorrects dans certaines conditions, obligeant Intel à rappeler les Pentium 60 Mhz à 100 Mhz vendus au grand public.

circuits spécialisés dits de "hauts niveau" tels les circuits et systèmes utilisables dans le domaine des télécommunications (téléphones cellulaires, protocoles de communications dont l'UMTS, etc ...), mais également dans les semi-conducteurs ou les circuits plus complexes, dits "système sur une puce" ("System on a Chip" ou SoC), intègrant des éléments mixtes matériels-logiciels réutilisables.

## 1.2 État de l'art

### 1.2.1 La vérification formelle

Dans toute la suite, nous limitons notre discussion sur la vérification formelle de circuits digitaux; les outils et techniques formelles dédiés à la vérification de logiciels et des compilateurs ne sont donc pas abordés.
Lorsque nous parlons de vérification formelle en milieu industriel, la plupart des personnes pensent à la vérification d'équivalence ("equivalence checking") ou à la vérification de modèles ("Model-checking"). En effet, bien qu'elles ne représentent pas toutes les catégories en vérification formelle, elles ont été introduites sur le marché sous forme d'outils commerciaux.
La *vérification formelle* [53] est une approche algorithmique de la vérification logique. Plus généralement, ce type de vérification s'associe à tout outil qui peut effectuer une preuve sur une description, de manière mathématique et exhaustive. Un outil de vérification formelle ne dit pas simplement si un problème existe mais aide aussi le concepteur à corriger le problème. En effet, le mot-clé ici est "prouver". Cela est différent de la simulation qui donne la réponse : "Je crois que cela fonctionne sur les cas qui ont été testés".
La complexité croissante des descriptions, la migration vers les systèmes sur une puce, et l'utilisation de blocs réutilisables IP ("Intellectual Property") ont contribué aux limitations de la simulation. La simulation sur ces descriptions (qui peuvent atteindre quelques millions de portes aujourd'hui), devient longue et coûteuse, sans compter la difficulté de trouver les vecteurs de tests nécessaires. De plus, elle n'est pas exhaustive et peut laisser échapper des erreurs [25].

#### La vérification d'équivalence

Les outils de *vérification d'équivalence* dominent actuellement le marché, principalement grâce à leur facilité d'utilisation. La vérification d'équivalence n'est pas simplement un substitut à la simulation fonctionnelle au niveau porte. Elle vérifie si deux circuits décrits dans un langage sont équivalents et répond à la question "Ai-je corrompu ma description en la modifiant ?". Ceci peut être utile si on possède déjà une représentation "vérifiée" d'un circuit et que l'on veut valider une nouvelle implémentation. Mais cela permet aussi de vérifier le bon déroulement d'une étape de synthèse.
Pratiquement, la vérification d'équivalence est utilisée à des niveaux variés d'implémentation : à partir du haut niveau RTL jusqu'au niveau "netlist" obtenue après placement-routage. Ainsi, elle sert à comparer deux descriptions, soit pour des conversions FPGA-ASIC (niveau porte/niveau porte), des réutilisations de descriptions (niveau porte/RTL), des migrations de langages (VHDL vers Verilog), des raffinements de descriptions (RTL/RTL), soit enfin pour des insertions de BIST

("Built-In Self Test") ou des remontées de fonctions à partir des transistors.
L'emploi de la vérification d'équivalence par rapport à la simulation est rentable lorsque la description dépasse les 200 000 portes.

Voici une liste non-exhaustive d'outils de vérification d'équivalence :

- Design Verifyer, vendu par Avant!
- Tuxedo, vendu par Verplex Systems
- Formality, vendu par Synopsys, Inc
- LEQ Logic Equivalency, vendu par Formalized Design
- Heck, vendu par Cadence Design Systems

### La vérification de modèle ("Model-checking")

Les outils de *Model Checking* [59, 22] veulent apporter une réponse à une question plus délicate : "Ai-je écrit la description que je voulais?". Cette technique requiert de l'utilisateur qu'il crée une *spécification* en entrant des propriétés sur sa description. Ce type de vérification permet de déceler des failles, comme les "deadlocks" (état qui bloquerait le système) ou des "exclusions mutuelles" (un système ne fonctionne pas avec un autre). Par exemple, une spécification peut être créée pour vérifier qu'un contrôleur d'un distributeur automatique de boissons rend bien la monnaie correctement et délivre bien la boisson demandée.

Le *Model Checking* est une technique qui consiste à vérifier les systèmes dont le comportement se modélise par des machines d'états finis tels que les circuits séquentiels ou les protocoles de communications. Le Model Checking symbolique possède ses racines dans les premiers concepts du Model Checking temporel, proposé par Clarke et Emerson en 1981. Dans le Model Checking temporel, les spécifications sont exprimées dans une logique appellé CTL (Computation Tree Logic), et les descriptions à valider sont modélisées en systèmes états-transitions. Une procédure efficace basée sur le calcul du point fixe est utilisée pour déterminer automatiquement si les spécifications correspondent aux systèmes états-transitions. Soit l'outil de Model Checking s'arrête avec la réponse "True", indiquant que la description satisfait la spécification, soit l'outil retourne "False". Dans ce dernier cas, l'outil produit un contre-exemple explicite. L'inconvénient de cette approche d'origine est le problème de *l'explosion combinatoire* : l'espace d'état des descriptions du monde industriel est typiquement trop grand pour permettre une recherche exhaustive.
En 1986, Bryant [20] a développé une nouvelle représentation des fonctions booléennes, appellées OBDDs (Ordered Binary Decision Diagrams). La contribution cruciale de Bryant a été de montrer qu'en fixant un ordre dans le test des variables booléennes présentes dans les diagrammes de décisions binaires, ces diagrammes devenaient des représentations canoniques. McMillan réalisa alors que les OBDDs peuvent représenter symboliquement les espaces d'états dans les systèmes états-transitions, et, en 1987, il développa un logiciel appellé SMV (Symbolic Model Verifier)

dans lequel les systèmes contenant $10^{20}$ états pouvaient être vérifiés, la technique du *model checking symbolique* était née. Actuellement, de nombreux travaux de recherche tendent à améliorer cette technique afin d'y incorporer de nouvelles techniques d'abstraction [27, 60, 7, 21].

Actuellement, les model-checking symboliques sont largements utilisés dans les industries des semi-conducteurs, SMV étant l'un des plus utilisé. Plusieurs compagnies ont intégrés les model-checking symboliques dans leurs propres outils de conception propriétaires. Par exemple, RuleBase développé au IBM Haifa Research Lab est utilisé dans des projets consistant à vérifier des protocoles de bus ou d'autres composants. Le model ckecker SVE développé à Siemens est appliqué dans de nombreux projets de développements internes, et a été commercialisé à des clients externes. D'autres outils commerciaux, basés sur le model-checking symbolique, incluent FormalCheck de Lucent, commercialisé par Cadence et Design Insight, vendu par Avant!.

### La démonstration de théorèmes

La *démonstration de théorèmes* [24] est une technique en marge des précédentes. Elle est peu utilisée mais suscite de plus en plus d'intérêt. La technique de démonstration de théorèmes est de nature plus mathématique que les techniques vues précédemment et, en fait, s'utilise pour résoudre des problèmes différents.

Actuellement, une description est souvent définie comme une machine qui va réagir en réponse à un stimulus. Avec la démonstration de théorèmes, il s'agit de formaliser le circuit dans une logique mathématique. En le définissant à un tel niveau d'abstraction, nous le convertissons complètement dans une approche formelle. Par exemple, cela fournit le moyen de créer une *spécification mathématique* sur la manière dont un multiplieur fonctionne, et d'être capable de le prouver. Cette approche permet ainsi des raisonnements arithmétiques qui peuvent être impossibles avec les autres techniques.
La démonstration de théorèmes a pour réputation d'être la technique la plus rigoureuse (les autres outils sont parfois implémentés sur des techniques non vérifiée) mais également la plus longue et la plus complexe. Il existe des outils commerciaux, citons

- GSVT, de Greentech Computing
- Lambda, de Abstract Inc (ayant fait faillite)
- NP Tools, de Prover Technology
- ProofPower, d'ICL Secure Systems (basé sur le démonstrateur HOL)

Mais ces outils ont peu de succès, car ils ne sont pas facilement manipulables. Il existe également des dizaines de prototypes universitaires et industriels, la plupart étant en licence publique gratuite pour les utilisateurs. Les plus connus sont :

- PVS [67], développé par SRI, est un démonstrateur inductif basé sur une logique d'ordre supérieur. Il dispose de nombreuses règles de déduction et est utilisé dans l'industrie. La démonstration n'est pas automatique et doit être guidée par l'utilisateur.

- ACL2 [49, 48] développé à CLI (Computational Logic Inc, Austin, Texas), puis à l'Université du Texas à Austin par J Moore et Matt Kaufmann, est un démonstrateur inductif basé sur une logique du premier ordre, sans quantificateurs. Il s'agit d'un démonstrateur automatique qui utilise un moteur d'enchaînement de règles de déduction. Cet outil est le successeur du démonstrateur de Boyer-Moore, Nqthm [16], afin de le rendre compatible avec un usage industriel sur de gros examples.

- Coq [6], développé à l'INRIA, est un démonstrateur de théorème inductif, basé sur une logique d'ordre supérieur. Plus précisément, cet outil est basé sur le calcul des constructions inductif, c'est-à-dire permet l'expression des démonstrations comme des termes, et donc leur traçabilité : ces démonstrations peuvent être revérifiées par un système indépendant et la correction d'un système ne repose que sur celle d'un noyau très réduit. Coq propose donc une intégration du raisonnement et du calcul.

- HOL [39], développé initialement par l'Université de Cambridge, est un démonstrateur inductif basé sur une logique d'ordre supérieur. Cette logique est simple mais expressive. La démonstration d'un théorème est manuelle. HOL a été utilisé pour la validation de nombreux systèmes matériels, logiciels ainsi que des protocoles de communication.

Un ingénieur de conception doit avoir de longs mois de formation pour utiliser un démonstrateur de théorèmes. Seule une poignée de gros industriels emploient des spécialistes pour vérifier par exemple des propriétés difficiles telles que le bon fonctionement des parties arithmétiques flottantes de leurs processeurs [43, 74].

### La simulation symbolique

La *simulation symbolique* n'est pas une technique de vérification nouvelle, elle a été utilisée depuis les années 70 par des fabricants de semiconducteurs. Mais ce n'est que récemment que des outils commerciaux sont devenus disponibles. Citons par exemple l'outil ESP d'Innologic Systems effectuant une simulation symbolique sur les descriptions écrites en Verilog.

La simulation symbolique est une simulation dirigée par évènements qui associe un symbole ou une variable comme valeur d'entrée. En effet, au lieu de prendre par exemple un vecteur de bits en entrée d'un simulateur, la simulation symbolique abstrait toutes les valeurs du vecteur en un symbole (par exemple V) et propage comme résultat les expressions logiques obtenues. Nous avons ainsi une relation entre les sorties et les entrées, quelles que soient les valeurs du vecteur.
La simulation symbolique fut utilisée pour la vérification de microprocesseur ou de protocoles ([23, 33, 8, 26]) mais des techniques hybrides utilisant cette technique suscitent de plus en plus d'intérêts ([76, 57, 2, 88, 65, 64, 71].

**Notre contribution**

Les démonstrateurs de théorèmes souffrent d'un manque d'automatisme et d'un manque de méthodes standards pour la modélisation et la preuve de systèmes matériels [1]. En effet, les industriels ont un réel intérêt pour les outils automatiques et facilement manipulables.
Nous nous proposons, dans cette thèse, d'apporter un environnement formel, utilisant démonstration de théorèmes et simulation symbolique, permettant la vérification d'un certain type de descriptions écrites en HDL ("Hardware Descriptions Languages").

Plus précisément, nous nous proposons :

- de fournir une méthode de formalisation de la sémantique d'un langage de description de matériel (HDL) vers un démonstrateur de théorèmes.
- de fournir un traducteur HDL $->$ démonstrateur,
- d'apporter des méthodes de preuves,
- de rendre utilisable ce modèle au moyen d'une interface utilisateur, facilement manipulable, générant des théorèmes intermédiaires et permettant des simulations numériques et symboliques.

Nous choisissons le langage de matériel VHDL [44, 4], langage riche, largement utilisé dans l'industrie et supporté par de nombreux outils commerciaux (Cadence, Synopsys, Mentor...). De plus, ce langage dispose d'une norme standard, ce qui permet de réaliser des outils conformes à sa sémantique.

Pour réaliser cet objectif, nous choisissons le démonstrateur Acl2. Celui-ci dispose d'une logique de premier ordre, sans quantificateurs et d'un moteur d'enchaînements de règles, ce qui le prédispose à une bonne automatisation. Mais ce qui le distingue avant tout de ses concurrents est son implémentation autour d'un compilateur Common Lisp [82, 83], dont il exploite un sous-ensemble de la syntaxe et sa rapidité d'exécution. De plus, Acl2 dispose d'une large gamme de bibliothèques arithmétiques, spécialement étudiées pour la formalisation et la preuve de systèmes matériels.

## 1.3 Organisation du mémoire

Le chapitre 2 introduit le démonstrateur de théorème Acl2. La formalisation de la sémantique de VHDL dans la logique du démonstrateur est définie dans le chapitre 3.
La simulation symbolique est discutée dans les chapitres 4.
Le sujet de la preuve de propriétés sur des descriptions VHDL est abordé dans le chapitre 5, leur réutilisabilité dans le chapitre 6.

L'implémentation de cette méthode de formalisation est discutée dans le chapitre 7 où un logiciel prototype, effectuant la traduction et la manipulation des modèles Acl2 de VHDL, est

présenté.

Le chapitre 8 conclut et présente les perspectives de ce travail.

# Chapitre 2

# Introduction au démonstrateur de théorèmes Acl2

## 2.1 Préliminaires

Le rêve des machines qui raisonnent a toujours accompagné l'ordinateur depuis sa création. Leibniz l'a exprimé de la façon suivante :
*Si nous avions un certain langage exact... ou au moins un type d'écriture philosophique vraie, dans lequel les idées se réduisent à un type d'alphabet de pensée humaine, alors tout ce qui dérive rationnellement de ce qui est donné peut être trouvé par un type de calcul, de la même façon dont on résoud les problèmes arithmétiques et géométriques. – Leibniz (1646-1716).*
Le rêve de Leibniz d'un calcul de la pensée humaine est apparemment devenu vrai. Les bits du monde mathématique, c'est-à-dire le calcul propositionnel et l'arithmétique élémentaire, peuvent être décrits formellement par des axiomes écrits comme des formules dans une syntaxe fixée. Des règles sont proposées pour la génération de nouvelles formules à partir des anciennes– règles avec la propriété que les nouvelles formules sont vraies si les anciennes le sont. C'est la base de fonctionnement d'un *système formel*.
ACL2 (A Computational Logic for Applicative Common Lisp) [50, 49, 48] est un démonstrateur de théorèmes développé à l'Université du Texas à Austin par J Moore et Matt Kaufmann. Acl2 est à la fois un langage de programmation et un moteur de raisonnement pour ce langage. Nous décrivons dans la section suivante la syntaxe qu'il emploie. La section "Raisonnement" décrit le moteur de raisonnement. Il s'agit essentiellement ici de rappeler des parties importantes du démonstrateur. Cette introduction n'est pas exhaustive et le lecteur trouvera de plus amples précisions dans la documentation sur Acl2 [49, 48]. Quelques caractéristiques d'Acl2 :

- Il dispose d'un langage de programmation conventionnel connu,
- Le modèle ACL2 peut être exécuté à une vitesse proche d'un simulateur C,
- Il dispose d'une vaste bibliothèque de théorèmes réutilisables (arithmétiques, simplifications, ...)
- Il s'agit d'un démonstrateur inductif de premier ordre, automatique.

Acl2 a servi bon nombre d'exemples industriels avec grand succès dès sa création. En effet, Acl2 a été conçu afin de répondre aux attentes des utilisateurs de Nqthm [16] sur des projets ambitieux. Parmi les projets utlisant Acl2 nommons :

- La preuve du théorème d'incomplétude de Gödel [52],
- La vérification de la description au niveau porte du microprocesseur FM9001 [19, 85]
- La déroulement correct de l'algorithme de la division en nombre flottant sur le processeur AMD5k86 [54]
- La confrontation entre théorèmes et séquence de micro-code a servi de preuve pour le processeur de traitement du signal CAP de Motorola (également dans [19]).
- Validation des implémentations de l'addition, soustraction, multiplication, division et racine carrée en virgule flottante du processeur AMD Athlon [73]
- Validation de machines pipelinées [78, 56, 51].
- Simulation à grande vitesse au sein de Rockwell Collins de modèles de processeurs [86].

## 2.2 Programmation

### 2.2.1 Le langage

Le langage de programmation d'Acl2 est en fait une extension d'un sous-ensemble de Common Lisp [82, 83]. Ce sous-ensemble ne contient pas les caractéristiques de Common Lisp liées aux effets de bord, c'est-à-dire les variables globales et les modifications destructrices des données[1]. Ainsi, Acl2 utilise un sous-ensemble fonctionnel ou applicatif de Common Lisp.
Lisp est un des plus vieux langages de programmation qui est toujours en utilisation. Il s'agit d'un langage *fonctionnel*, c'est-à-dire composé essentiellement de *fonctions*. Une *fonction* est une application d'un domaine vers un ensemble de valeurs. L'application d'une fonction $f$ à un élément $n$ de son domaine de définition s'écrit en Lisp $(f\ n)$.
Les fonctions d'Acl2 sont totales : elles sont définies pour tout type de valeurs en entrée. Lorsqu'une fonction Acl2 est appliquée à des arguments qui sont en dehors du domaine correspondant à la fonction Common Lisp, une valeur particulière est alors calculée. Par exemple, l'addition de constantes non numériques telles que (+ t nil) n'est pas définie en Common Lisp (et donne une erreur dans la plupart des cas), mais dans Acl2 (+ t nil) est définie comme étant 0. Les *gardes* d'Acl2 permettent de caractériser le domaine attendu d'une fonction et de montrer également qu'elle est bien typée. Nous verrons dans le chapitre 3 que les gardes permettent d'exécuter les fonctions plus rapidement en relayant l'exécution au moteur Common Lisp[2].

1. certaines commandes "destructives" sont toutefois implémentées avec une "notion d'état". Elles prennent en argument et rendent comme valeur *l'état global d'Acl2*.
2. Le moteur Common Lisp étant le compilateur Lisp (gcl, Mcl, Allegro..) servant de base à Acl2

### 2.2.2 Les types de données

Acl2 supporte cinq types de données, illustrés ci-dessous :

- Les nombres (entiers, rationnels, complexes): 0, -123, 22/7, #c(2 3)
- Les caractères: `#\A, #\a, #\c, #\Space`
- Les chaines : “Ceci est une chaîne”
- Les symboles: nil, toto, PAQ1::func-x
- Les listes (1.2), (a b c), ((a .1) (b.2))

Une description informelle de leur construction logique peut être présentée ainsi :

- Pour les nombres : Les entiers positifs (les *naturels*) sont construits à partir de 0 au moyen d'une fonction de successeur. Les entiers négatifs sont construits à partir des naturels par la fonction négation. Les rationnels sont construits à partir des entiers par division. Les nombres complexes sont construits à partir de paires de rationnels.
- Pour les caractères: Un caractère est construit à partir d'un nombre naturel (plus petit que 256).
- Pour les chaines : Une chaine peut être définie comme une séquence finie de caractères.
- Pour les symboles : Un symbole peut être construit comme deux chaines : la première indiquant le paquetage Lisp de provenance et l'autre nommant le symbole à l'intérieur de ce paquetage . Rappelons que le symbole PAQ1::func-x signifie en Lisp : le symbole func-x dans le paquetage PAQ1. Le symbole toto étant une forme abrégé de Lisp::toto où Lisp est le paquetage de base.
- Pour les listes : Elles sont considérées comme des paires d'objets (l'objet étant en partie gauche est appellé le *car*[3] de la liste et le reste en partie droite le *cdr*[4]).

Les programmes Acl2 sont composés d'*expressions*, aussi appelés *termes*. Alors que la plupart des langages de programmation ont des expressions, des instructions, des blocs, des procédures, des modules, etc ... Acl2 ne possède que la notion d'expressions.
Une *expression* est :

- un symbole de variable (x, a, ....)
- un symbole de constante (t, nil ou un symbole déclaré avec la commande **defconst**)
- une expression constante ('a', “chaine”, 7, '(a b c)..)
- l'application d'une fonction, *f*, de *n* arguments, à *n* expressions, *a1, ..., an*, écrite (*f a1 ... an*).

3. Signifie historiquement, “Contents of the address register”
4. Signifie historiquement : “Contents of the decrement register”

Voici un exemple d'expression:

```
(if (equal date '(august 14 1989))
    ``Happy birthday, Acl2 !''
     nil)
```

Cette expression contient le symbole de variable date, le symbol de constante nil, deux expressions de constantes (une liste et une chaine), et deux applications de fonctions (les symboles de fonctions if et equal). La valeur de cette expression est dépendante de la valeur affectée à la variable `date`. Elle rend "Happy birthday, Acl2!" si `date` est la liste `(august 14 1989)`, le symbole `nil` sinon.
Les définitions de fonctions se font avec la commande **defun**. Exemple de définition :

```
(defun fact (n)
   (if (zp n)
       1
      (* n
        (fact (- n 1)))))
```

Cette expression définie `fact` comme une fonction qui prend un argument et retourne sa factorielle. Par exemple, l'évaluation de `fact 3` retourne 6. Dans cette définition, `(fact n)` rend 1 si `zp n` est vrai, c'est-à-dire si l'argument `n` n'est pas un entier positif. Dans le cas contraire, la fonction calcule en premier la factoriel de `n - 1`: `(fact (- n 1))`, et multiplie le résultat avec `n`. La fonction `(zp n)` est une fonction prédéfinie dans la logique d'Acl2 et qui est équivalente à :

```
(if (integerp n) (<= n 0) T)
```

La figure 2.1 montre certaines fonctions et constantes prédéfinies dans la logique d'Acl2. Il est important d'adopter des *schémas récursifs usuels* de définition de fonctions, afin de faciliter leur compréhension par Acl2 au niveau logique. L'utilisation de certains prédicats reconnaisseurs (zp, endp, atom) rend la fonction totale. Ainsi :

- Décrémentation d'un nombre naturel ((zp n) retourne vrai si n est 0 ou n'est pas un naturel, faux sinon):

```
(defun f (...n...)        ; f est récursif sur n
   (if (zp n)             ; si n est 0 (ou non naturel)
       <...>
       <... (f ... (- n 1) ...) ...>))  ; récursion sur n - 1
```

- Parcours des éléments d'une liste ((Endp x) retourne vrai si x n'est pas une liste) :

```
(defun f (...L...)   ; f est récursif sur la longueur d'une liste L
   (if (endp L)      ; si L est vide (ou n'est pas de type liste)
       <...>
       <... (car L) (f ... (cdr L) ...) ...>))
                     ; utilisation du (car L) et récursion sur (cdr L)
```

| ACL2 Functions and Constants | Informal Description | |
|---|---|---|
| `T` | True value. | * |
| `nil` | False value as well as the empty list. | * |
| `(+ x y)` | $x + y$ | * |
| `(- x y)` | $x - y$ | * |
| `(* x y)` | $x \times y$ | * |
| `(/ x y)` | $x/y$ | * |
| `(mod x y)` | $x$ **mod** $y$ | * |
| `(expt x y)` | $x^y$ | * |
| `(1+ x)` | $x + 1$ | * |
| `(1- x)` | $x - 1$ | * |
| `(< x y)` | $x < y$ | * |
| `(<= x y)` | $x \leq y$ | * |
| `(equal x y)` | $x$ equals $y$. | * |
| `(if x y z)` | If $x$ is true, returns y. Otherwise $z$. | * |
| `(not x)` | $\neg x$ | * |
| `(and x y)` | $x \wedge y$ | * |
| `(or x y)` | $x \vee y$ | * |
| `(implies x y)` | $x \rightarrow y$ | * |
| `(iff x y)` | $x \leftrightarrow y$ | * |
| `(car x)` | First element of cons pair $x$. | * |
| `(cdr x)` | Second element of cons pair $x$. | * |
| `(cadr x)` | `(car (cdr x))` | * |
| `(cddr x)` | `(cdr (cdr x))` | * |
| `(cons x y)` | Cons pair of $x$ and $y$. | * |
| `(null x)` | $x$ is *nil*, i.e., the empty list. | * |
| `(consp x)` | $x$ is a cons. Note `(consp nil)` is false. | * |
| `(endp x)` | $x$ is *nil* or an atomic object | * |
| `(len x)` | Length of list $x$. | |
| `(append x y)` | Concatenation of list $x$ and $y$. | * |
| `(nth n x)` | The $n$'th element of list $x$. | * |
| `(nthcdr n x)` | Removes the first $n$ elements from list $x$. | * |
| `(list x y ...)` | List whose elements are $x$, $y$, ... | * |
| `(integerp x)` | True if $x$ is an integer. | * |
| `(true-listp x)` | True if $x$ is a list terminating with *nil*. | |
| `(zp x)` | $x$ is not a positive integer. | |

FIG. 2.1 – *Description des fonctions et constantes prédéfinies. Celles qui sont marquées d'un astérisque sont définies en Common Lisp.*

### 2.2.3 La modélisation d'un système

L'activité consistant à modéliser le problème dans la syntaxe formelle d'une certaine logique est appellée *formalisation*. Les principales difficultés de l'utilisation d'un démonstrateur de théorèmes sont de trouver une manière efficace de modéliser un système et de formaliser ses propriétés. En effet, nous ne pouvons pas prouver qu'un circuit, un microprocesseur ou un autre système physique est correct. Nous pouvons seulement prouver qu'un certain *modèle* de ce système possède une certaine propriété. C'est à l'utilisateur d'utiliser la logique pour créer un modèle du système et ensuite d'écrire les propriétés qui expriment sa *compréhension* de celui-ci. Comme nous allons le voir dans la section qui suit, Acl2 dispose d'axiomes et de règles d'inférence établies de manière directe (théorèmes) ou indirecte (définition de fonctions) par l'utilisateur.

## 2.3 Raisonnement

### 2.3.1 Théorèmes et règles de déduction

Un *théorème* est soit un axiome, soit une formule produite par application successive de certaines règles de déduction (ou d'inférence) à d'autres théorèmes. Une *preuve* est une structure finie montrant la dérivation d'un théorème à partir des axiomes. *Un théorème est toujours vrai.* L'argument de cette propriété fondamentale est le suivant :

1. Les axiomes sont toujours vrais.
2. les règles d'inférence préservent cette propriété.
3. Puisqu'un théorème est dérivé par application d'un nombre fini de règles d'inférence aux axiomes, les théorèmes doivent toujours être vrais.

Pour prouver qu'une formule est un **théorème**, Acl2 applique une stratégie basée sur l'application des axiomes et des règles existantes. Si la preuve marche, Acl2 ajoute le théorème dans sa base de données, et cette nouvelle règle change la future stratégie du démonstrateur de théorèmes. Si la preuve échoue, il faut soit changer le *prétendu* théorème, soit ajouter un guide sous forme d'un nouveau théorème *intermédiaire* qui sera transformé en nouvelle règle d'inférence; ce dernier théorème n'est pas la propriété souhaitée mais une *information supplémentaire* nécessaire au démonstrateur pour prouver la propriété.

Un théorème à prouver se défini comme suit :

```
(defthm nom_du_théorème
        hypothèse
        conclusion)
```

Voici un exemple simple. Supposons que nous voulons prouver le théorème suivant :
$\forall n, 3 + (1 + N) = 4 + N$
Formulons le théorème à prouver :

```
(defthm theorem_a_prouver          ; nom du théorème
```

```
    (implies (integerp N)          ; N entier
             (equal (+ 3 (+ 1 N))
                    (+ 4 N))))
```

Acl2 a besoin d'une règle de propriété (associativité) pour prouver le théorème. Il faut donc rajouter une information au démonstrateur. Le théorème *intermédiaire* à prouver serait de la forme :

```
(defthm intermediate_lemma1          ; nom du théorème
    (implies (and (integerp a)       ; si a,b et c sont des entiers
                  (integerp b)
                  (integerp c))
             (equal (+ a (+ b c))
                    (+ (+ a b) c))))
```

Nous donnons ici la forme la plus générale de la propriété, afin que celle-ci puisse être utilisée dans la plupart des cas. Ainsi, Acl2 gagne l'information de cette décomposition et ajoute une nouvelle règle d'inférence qui se manifeste par une règle dite de *réécriture*. Cette règle dit que si les hypothèses (a, b et c entiers) sont vérifiées sur la formule en cours de vérification par Acl2, alors toute expression de la forme `(+ a (+ b c))` sera remplacée par la forme `(+ (+ a b) c)`. Par cette nouvelle règle, Acl2 remplace `(+ 3 (+ 1 N))` par `(+ (+ 3 1) N)` et va évaluer automatiquement l'addition de 3 avec 1, ce qui donne `(+ 4 N)`.

Suivant les cas, le théorème initial (appelé "But") peut se diviser en "sous-buts", ayant la même conclusion mais des hypothèses distinguant des cas plus précis. Par exemple en donnant comme hypothèse `(integerp x)`, Acl2 va former deux sous-buts, l'un avec l'hypothèse `(and (integerp x) (<= x 0))` et le second avec l'hypothèse
`(and (integerp x) (>= x 0))`. Mais c'est plus particulièrement lorsqu'il existe des instructions conditionnelles ("if") que Acl2 décompose en sous-buts: (nous supposons c, f, g et h définies)

```
(defthm theoreme_a_prouver
   (equal (f x)
          (if (c x)
              (g x)
              (h x))))
```

se décompose en deux sous-buts :

```
(defthm sous-but1-theoreme_a_prouver
  (implies (c x)
           (equal (f x)
                  (g x))))

(defthm sous-but2-theoreme_a_prouver
  (implies (not (c x))
           (equal (f x)
                  (h x))))
```

### 2.3.2 Le principe de définition

Le principe de définition permet d'ajouter de nouveaux axiomes par la définition de nouveaux symboles de fonction (grâce à la commande **defun** vue dans la partie Programmation), à condition que la cohérence de la logique soit préservée.

Le *principe de définition* permet ainsi d'introduire une extension au système formel actuel.
*Principe de définition.* La définition (`defun` *f* ($x_1$ ... $x_n$) *body*) est *admissible* si et seulement si :

- *f* est un nouveau symbole de fonction, c'est-à-dire qu'il n'existe pas un symbole de même nom déjà existant dans le système formel;
- les $x_i$ sont des symboles de variables distincts;
- *body* est une expression (cf partie Programmation), pouvant utiliser récursivement *f*;
- la définition doit posséder un argument disposant d'une *mesure* qui décroit pour les appels récursifs. Cette mesure doit être discrète et bornée inférieurement, pour garantir que la récursion se termine.

Si la nouvelle définition est admissible, l'effet logique est d'ajouter un nouvel axiome :

- *Axiome de définition pour f*: $(fx_1...x_n) = body$

Lors d'un appel récursif, une *mesure* ordinale (voir rappel ci-dessous) des $x_i$ doit décroître dans chaque appel récursif. En pratique, l'utilisateur identifie un terme représentant la mesure, $m$, et le fournit, la plupart du temps implicitement, lors de la définition de la fonction. Exemple, lors de la définition récursive de la factorielle de $n$ :

```
(defun fact (n)
   (if (zp n)              ; Si n est 0 (ou non-naturel)
       1                   ; retourne 1;
      (* n                 ; sinon multiplie n par
        (fact (- n 1)))))  ; fact de n-1.
```

la mesure est la valeur de $n$ car elle décroit à chaque appel de la fonction $fact(n-1)$.
Nous rappelons ci-dessous la définition des nombres ordinaux :

#### Les nombres ordinaux

Considérons les entiers naturels, c'est-à-dire 0,1,2,.... L'ensemble des entiers naturels est muni de la relation d'ordre total "<" :$0 < 1 < 2 < ...$, c'est-à-dire si nous prenons deux nombres distincts $x$ et $y$, nous avons nécessairement un plus petit que l'autre. De plus, l'ensemble des naturels est infini et suivant cette relation d'ordre, il ne possède pas un plus grand élément et il est bien fondé. Que cet ensemble soit bien fondé signifie que l'on ne peut y trouver de chaînes infiniment décroissantes. En d'autres termes, si l'on part d'un entier positif quelconque, on ne peut le diminuer régulièrement sans finir par aboutir sur 0, le plus petit des entiers naturels.
Les ordinaux sont une extension des nombres naturels. L'idée est d'ajouter de "nouveaux" nombres après les naturels. Par exemple, si nous ajoutons $\omega$, nous avons $0, 1, 2, ..., \omega$, où $0 < 1 < 2 < ... < \omega$. Notons que ce nouvel ensemble étend les naturels, est dénombrable, contient

un plus grand élément, est linéairement ordonné et est bien fondé. Les ordinaux sont tous les éléments d'un ensemble qui contient une extention. Les premiers ordinaux sont par exemple $0, 1, 2, ..., \omega, \omega + 1, ..., \omega \star 2$. En continuant d'ajouter des nombres, nous obtenons $\omega, \omega \star 2, \omega^2, ...., \omega^\omega, ..., \omega^{\omega^{\omega^\omega}}, ...\epsilon_0$. Les ordinaux d'Acl2 correspondent à l'ensemble des ordinaux plus petits que $\epsilon_0$[5]. Acl2 code les ordinaux sous forme de liste, par exemple:
$\omega^2 + (\omega \star 4) + 3$ se code : (2 1 1 1 1 . 3)
$\omega^\omega + \omega^{99} + (\omega \star 4) + 3$ se code : ((1.0) 99 1 1 1 1.3)
Les fonctions suivantes sont prédéfinies dans Acl2 et accompagnent la formalisation des ordinaux :

`(e0-ordinalp x)` retourne *t* ou *nil* suivant que l'argument x est un ordinal ou non

`(e0_ord-< x y)` retourne *t* ou *nil* suivant si x représente un ordinal inférieur à y.

`(acl2-count x)` retourne un ordinal qui mesure la "taille" de l'object x.

### 2.3.3 Le principe d'induction

Dans la logique traditionnelle du premier ordre, le *principe d'induction* est souvent défini comme suit: "Pour prouver que $\forall n \phi(n)$, il est suffisant de prouver (a) $\phi(0)$ et (b) $\forall n(\phi(n) \rightarrow \phi(n+1))$." Ici, la variable $n$ appartient à l'ensemble des nombres naturels. L'obligation (a) est appelée le *cas de base*; l'obligation (b) est appelée le *pas d'induction*. L'hypothèse de l'implication dans (b) est appelée l'*hypothèse d'induction* et sa conclusion est appelée *conclusion d'induction*. Notons que le principe d'induction formalisé ainsi est une règle syntaxique qui décrit comment générer un nombre fini de formules, ici juste deux, pour prouver un nombre infini de cas (pour toutes les valeurs de $n$).

Dans le cas des autres types de données, la fonction `acl2-count` extrait la correspondance avec les entiers, par exemple, la fonction `acl2-count` appliquée à une liste donne la longueur de celle-ci.

Les ordinaux permettent de raisonner sur des structures complexes, telles que, par exemple, des listes de listes, c'est-à-dire des structures où il y a une "double mesure".

### 2.3.4 Le principe d'encapsulation

Acl2 fournit un autre principe d'extension (*c.à.d* principe qui permet la définition de nouvelles règles d'inférence par l'utilisateur) que la définition de nouvelles fonctions : Il s'agit du principe d'*encapsulation*. Ce principe permet l'introduction de nouveaux symboles de fonctions non-définis mais possédant des propriétés. Des théorèmes doivent être prouvés pour établir que les propriétés sont satisfaites. Par exemple, l'encapsulation suivante introduit une propriété (appelée aussi *contrainte*) sur le nouveau symbole de fonction *f*.

```
(encapsulate (((f x1 ... xn) => *)) ; arité de la fonction
```

5. $\epsilon_0$ est le dernier ordinal $o$ tel que $\omega^o = o$

```
(local (defun f (x1 ... xn) body))
(defthm constraint φ))
```

La commande **encapsulate** ci-dessus permet l'introduction de la fonction $f$ avec la précision de son arité. La commande *local* signifie que ce qui suit n'est pas visible depuis l'exterieur de la commande. Donc, en donnant une définition à la fonction $f$ en local, celle-ci permet d'admettre le théorème *constraint*. Une fois la forme **defun** et la forme **defthm** admises, Acl2 étend le système formel en rajoutant :

Axiome : $f$ possède la propriété $\phi$.

Nous noterons dans cette axiome que $f$ n'est pas définie, cependant l'admissibilité de l'encapsulation assure qu'il existe au moins une fonction (celle définie en *local* dans l'encapsulation) qui satisfait la propriété $\phi$.

Par exemple, supposons que nous voulons déclarer une fonction *func*, prenant deux argument $x$ et $y$, qui possède la propriété d'être commutative, associative et la contrainte supplémentaire que sa valeur de retour est positive. Voici la déclaration :

```
(encapsulate (((func * *) => *))

 ;;;;; Definition d'une fonction ``témoin'' non visible
 ;;;;; de l'extérieur
   (local (defun func (x y)
               (+ (abs x)
                  (abs y))))

 ;;;;;;;;;; Propriété de commutativité
  (defthm commutativite_de_func
     (implies (and (integerp x)
                   (integerp y))
         (equal (func x y)
                (func y x))))

 ;;;;;;;;;; Propriété d'associativité
  (defthm associativite_de_func
     (implies (and (integerp x)
                   (integerp y)
                   (integerp z))
         (equal (func x (func y z))
                (func (func x y) z))))

 ;;;;;;;;  Contrainte : la valeur de retour est un entier positif
 (defthm valeur_de_func
   (implies (and (integerp x)
                 (integerp y))
     (and (integerp (func x y))
```

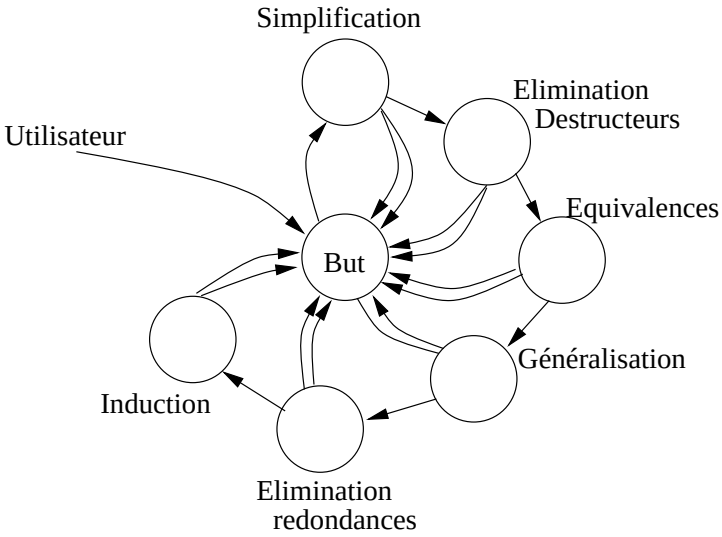


FIG. 2.2 – *Organisation du démonstrateur de théorèmes*

```
(>= (func x y) 0)))))
```

Le principe d'encapsulation est très utile pour définir une classe de fonctions possédant toutes la même propriété. En effet, tout théorème prouvé sur $f$ est alors valide pour toutes les fonctions $\hat{f}$ possédant la propriété $\phi$.
Supposons que $\hat{\psi}$ soit la formule obtenue à partir de $\psi$ par le remplacement du symbole $f$ par $\hat{f}$[6]. Alors une règle d'inférence, appelée l'*instanciation fonctionnelle* [15], dit que à partir de tout théorème $\theta$ on peut inférer le théorème $\hat{\theta}$ pourvu que $\hat{\phi}$ soit un théorème.

Nous allons nous servir du principe d'encapsulation pour raisonner sur des composants VHDL ayant la même propriété. L'avantage étant de s'abstraire de la manière dont le composant est écrit (degré d'abstraction, ...). De plus, le remplacement d'un composant (ou d'une fonction) plus simple et plus rapide, possédant la même propriété, nous permet alors d'accélérer la simulation et les preuves ultérieures du circuit contenant ce composant [79].

### 2.3.5 Fonctionnement du démonstrateur de théorèmes

#### La tactique de preuve par défaut

Pour tenter de démontrer qu'une formule est un théorème, Acl2 dispose de deux ensembles de règles: celles émises par l'utilisateur et celles disponibles déjà chargées dans la base de données d'Acl2. Pour montrer un nouveau théorème, le démonstrateur utilise une stratégie particulière, représentée sur la figure 2.2. La démonstration se déroule par chainage avant c'est-à-dire que chaque heuristique transforme la formule de départ en un ensemble de sous-buts à démontrer pour prouver le but initial. Si une heuristique ne s'applique pas à la formule, cette dernière demeure inchangée, et ainsi de suite jusqu'à ce qu'il n'y ait plus de buts à démontrer (la preuve à réussi) ou que l'on tombe sur une condition d'échec (la preuve a échoué).
Les deux premières heuristiques visent à simplifier autant que possible la formule de départ. Les

6. $\hat{f}$ doit posséder la même arité que $f$

trois suivantes servent de préparation à une preuve par induction de la formule qui est alors aussi simple et générale que possible (au sens des heuristiques employées).

Il est toutefois possible de modifier la tactique de preuve par défaut en ajoutant des *hints*[7].

#### Les types de règles d'inférence

Par défaut, la nouvelle règle créée lors de la preuve d'un nouveau théorème est une règle de réécriture. Cependant l'utilisateur a la possibilité de choisir le type de règle généré, ceci afin qu'il soit employé de manière plus efficace.

La commande pour ajouter une nouvelle règle d'un type donné *class* est :

```
(defthm nom formule
:rule-classes (class))
```

Le lecteur trouvera en annexe la liste des classes de règles d'Acl2.

### 2.3.6 Interaction avec l'utilisateur

Acl2 se présente sous la forme d'une interface textuelle avec un indicateur : `ACL2 !>`. Lorsqu'une commande est tapée, Acl2 affiche son évaluation (et éventuellement des commentaires) et réaffiche `ACL2 !`. Cette boucle d'évaluation appelée *read-eval-print loop* sert également à définir des fonctions et des théorèmes.

Par exemple, l'évaluation de :

```
(+ 4 (* 3 2))
```

affiche : `10`

Lorsque l'on définit une fonction ou un théorème, Acl2 affiche trois types de commentaires :

- La justification de l'acceptation de la fonction ou théorème (la sortie de la preuve)
- Les règles employées pour ces preuves
- QED ou Failed avec le temps affiché en secondes.

Un exemple de session est présenté en Annexe.

7. guide donné explicitement lors de l'écriture d'un théorème

# Chapitre 3

# Formalisation de la sémantique de VHDL dans Acl2

## 3.1 Etat de l'art

Cette section adopte la terminologie de VHDL, et suppose que le lecteur est familier avec les concepts essentiels de ce langage.
Le langage VHDL a fait l'objet de nombreuses modélisations, dûes notamment à une sémantique en terme de simulation définie de manière non rigoureuse [44]. Nous énumérons quelques approches récentes et qui adoptent une approche fonctionnelle ou opérationnelle. Cette approche de sous-ensemble de VHDL est dédiée à la réalisation de preuves de propriétés ou d'équivalence sur le langage ou sur un simulateur abstrait du langage. Ces preuves sont effectuées à l'aide de démonstrateurs de théorèmes ou d'assistants de preuves.

- Ainsi, dans [18], une sémantique fonctionnelle de VHDL'87 est présentée pour les descriptions comportementales. Elle ne considère que les signaux et seuls les temps physiques sont représentés. Le délai $\delta$ est interdit, cela signifie que les affectations à délai $\delta$ et les temporisations générant des $\delta$-cycle (`wait for 0 ns`) ne sont pas tolérées. Les fonctions de résolution ne sont pas traitées: un signal est affecté dans un seul process. Cette sémantique est ensuite mise en oeuvre dans la logique de Hoare. Un système de validation, écrit en Prolog, réduit les preuves sur des programmes VHDL aux preuves sur la validité des conditions initiales.

- Dans [75], Russinoff présente une sémantique informelle d'un sous-ensemble structurel/flot de données asynchrone de VHDL dans la logique du démonstrateur Nqthm. Cette modélisation repose sur la représentation du modèle du temps de VHDL comme un couple d'entiers naturels dont la première composante représente le temps physique et la seconde le temps delta, et la représentation des signaux comme des listes d'évenements qui se produisent sur les signaux. Un événement est un couple $(v,t)$ signifiant qu'un signal prend la valeur $v$ au temps $t$ de simulation. La caractéristique de cette formulation est d'avoir été implémentée dans le démonstrateur Nqthm et d'avoir permis la preuve de propriétés comportementales sur des circuits combinatoires et séquentiels simples. Le sous-ensemble

VHDL traité est cependant limité : seule l'affectation concurrente de signal et la gestion des composants sont considérées.

- Read et Edwards, dans [69], présentent une sémantique fonctionnelle exécutable d'un sous-ensemble VHDL appelé VHDL- dans le démonstrateur Nqthm. Ce sous-ensemble comprend les variables et les signaux. Les instructions concurrentes considérées sont l'instanciation de composants, l'affectation concurrente de signal et le process. Les sous-programmes sont inclus dans VHDL-. De plus, aussi bien le temps physique que le temps $\delta$ sont pris en compte. Ce moteur de simulation est exécutable, ce qui a permis la comparaison des résultats obtenus avec ceux de simulateurs commerciaux tels ceux de Cadence. En revanche, l'aspect preuve formelle n'est pas abordé du fait de la difficulté de raisonner sur ce simulateur formel.

- Nicoli, dans [66], propose également un simulateur formel d'un sous-ensemble VHDL proche des sous-ensembles VHDL synthétisables. Ce sous-ensemble reprend celui de [69] en lui ajoutant les instructions "wait" (sans la clause "for"), des types de données plus riches (cependant sans formaliser le type des tableaux utilisé pour la représentation des vecteurs de bits). En revanche, la mise en application de ce simulateur par rapport à une description donnée est difficile et faite manuellement. Les preuves se basent sur les sorties de simulation, et, bien qu'intéressantes, elles se font sur une formalisation complexe basé sur Nqthm.

- Dans [13], un sous-ensemble synchrone de VHDL, appelée P-VHDL, est assez complet, il contient :

  - les signaux,
  - les instructions concurrentes : l'affectation concurrente de signal, une version simplifiée de l'affectation conditionnelle de signal, les process avec liste de sensibilité, les process de synchronisation (comportant en première instruction un "wait" sur l'horloge de synchronisation)
  - les instructions séquentielles (en nombre limité): `if`, affectations de signaux et de variables, la séquence d'instructions.

  Cette sémantique ne considère pas le temps physique, de plus le mécanisme de simulation est présenté de manière informelle sous forme algorithmique. Son originalité réside dans le fait qu'elle modélise la phase d'élaboration. Cette sémantique n'a pas été implémentée.

Nous proposons dans la suite de nous focaliser sur un sous-ensemble synchrone de VHDL, qui ne contient pas de caractéristiques spécifiques par rapport aux autres définitions sémantiques. Ce sous-ensemble comprend les signaux, les variables et les paramètres génériques. Les instructions concurrentes considérées sont l'instanciation de composants, l'affectation concurrente de signaux et le process. Les instructions séquentielles sont le `if`, les affectations de signaux et de variables, les structures `case`, les boucles `for...loop`, la séquence d'instructions. Les fonctions VHDL sontégalement traités. Nous proposons pour

ce sous-ensemble un environnement de développement qui donne la possibilité de valider les descriptions par une approche mixte de simulation numérique, simulation symbolique, et preuve formelle, au travers d'une interface utilisateur.

## 3.2 Le sous-ensemble VHDL

La nécessité de se restreindre à un sous-ensemble de VHDL provient de sa complexité et de la difficulté d'exprimer toutes ses caractéristiques. Ce langage est complexe de par la richesse de ses unités syntaxiques, de ses aspects concurrents et séquentiels mais aussi de sa sémantique exprimée en terme de simulation dirigée par évenements. Nous souhaitons effectuer la preuve de systèmes matériels (circuits, processeurs, etc ..) décrits à un haut niveau d'abstraction. Pour réaliser cet objectif, nous nous focalisons donc sur un "*sous-ensemble synthétisable*" de VHDL, actuellement en cours de standardisation [45]. Ce sous-ensemble exclut le temps physique et les types non-discrets. Nous nous limitons à une synchronisation à une seule horloge et n'acceptons pas les process où l'horloge et des signaux de types "set" et "reset" sont présents à l'intérieur d'une même liste de sensibilité.

Un circuit est décrit par une **entité**, qui déclare ses signaux d'interface (les directions reconnues étant **in**, **out**). Au plus un de ces signaux est l'horloge principale. Au moins une **architecture** est associée avec l'entité, et décrit le comportement du circuit. À l'intérieur de l'architecture, les processus concurrents (*process*), les instructions concurrentes et les instanciations de composants peuvent communiquer entre eux par l'intermédiaire de signaux déclarés localement à l'architecture; pour garantir l'aspect déterministe du comportement, l'usage des variables partagées ("shared variables") est exclu du sous-ensemble synthétisable, c'est-à-dire que les variables ne peuvent être déclarées qu'à l'intérieur des *process*.

*Nous considérons des descriptions VHDL constituées d'une entité et d'une architecture, de plus, nous associons la notion de composant avec un couple entité-architecture.*

De manière informelle, les élements syntaxiques qui apparaissent dans cette description sont:

- les objets: les constantes, les signaux, les variables, les paramètres génériques. Des valeurs initiales peuvent être précisées pour ces trois dernières classes d'objets.
- les types de données basiques utilisables :
  - Les entiers
  - Les bits
  - Les vecteurs de bits de taille fixée
  - Les types *signed* et *unsigned*
- les fonctions: nous autorisons l'utilisation des fonctions avec les types de données ci-dessus.

- les entités: elles introduisent la déclaration de ports in/out, éventuellement avec initialisations.

- les architectures: elles sont constituées de déclarations de signaux, déclarations de composants, et d'instructions concurrentes.

- les instructions process: elles incluent les déclarations de variables et les instructions séquentielles. Nous supposons que tous les process sont écrits dans une forme **normale**, c'est-à-dire avec une seule instruction **wait** (voir [34] pour une justification théorique) : **wait until** `<front-d'horloge>`, où `<front d'horloge>` est défini dans le standard pour RTL synthétisable [45]. Dans toute la suite, nous ajoutons la restriction que tous les process sont synchronisés sur le même front d'horloge;

- les instructions séquentielles: on ne considère que les instructions simples :

  - l'affectation de variable;
  - l'affectation de signal sans notion de temps (à retard nul).
  - l'instruction conditionnelle `if ...then...else...` la plus simple. L'instruction VHDL `case` peut se ramener à une ou plusieurs séquences d'instruction `if`.
  - la boucle `for...loop` contenant une condition d'arrêt.

- instructions concurrentes Nous considérons les instructions process, les affectations de signaux et les instanciations de composants.
  D'après la définition VHDL, toutes les instructions concurrentes dans une architecture (affectations de signaux, assertions, appels de procédures, blocs) sont traduites en "process" équivalents. Cependant, ces process équivalents ne sont pas synchronisés par l'horloge et peuvent introduire des $\delta$-cycles entre deux cycles d'horloge. La solution est donc d'exécuter *stab* fois ces instructions concurrentes, $stab >= 1$ (avec les mises à jour des pilotes). Cette valeur représente le nombre de $\delta$-cycles nécessaires pour stabiliser les instructions concurrentes. Cette valeur est calculée automatiquement à partir de la liste des instructions concurrentes.

- les expressions: nous considérons les expressions arithmétiques et booléennes.

  - les opérateurs arithmétiques élémentaires, ainsi que la valeur absolue sur les entiers.
  - les opérateurs booléens
  - les opérateurs de comparaison
  - les opérateurs agissant sur les vecteurs (longueur, reverse, etc ...)

- les fonctions VHDL: nous considérons les fonctions non récursives, ne comportant pas d'instructions exit.

L'usage des types *résolus* n'est pas implémenté actuellement.

## 3.3 VHDL : Un langage de simulation

Nous allons présenter le mécanisme de simulation de VHDL restreint au sous-ensemble du paragraphe précédent. Les signaux sont utilisés dans la pratique pour modéliser des bus, des latches, des registres ... Ils sont modifiés à l'aide de l'instruction d'affectation de signal (<=), cette instruction ne change pas la valeur courante (au temps courant de simulation) mais ses valeurs futures. Le couple constitué d'une valeur et du temps auquel un signal va prendre cette valeur est appelé une **transaction**. A un même signal peuvent être associées plusieurs transactions qui sont regroupeés dans une même structure appelée **pilote**. Les transactions sont ordonnées par composantes temporelles croissantes dans un pilote. Pour un pilote donné, il existe une unique transaction dont la composante est inférieure ou égale au temps courant de simulation, cette valeur est appelée **la valeur courante** du pilote. En début de cycle de simulation, si le temps de simulation est égal à la composante temporelle de la deuxième transaction d'un pilote, le signal associé est dit **actif** : dans ce cas, la première transaction du pilote est effacée et la deuxième devient la nouvelle valeur courante du pilote. Dans le cas contraire, le pilote reste inchangé.
Un **pilote** est une source d'un signal dans la mesure où, en début de chaque cycle de simulation, le simulateur consulte le pilote d'un signal pour donner une valeur à ce dernier; il s'agit de la **valeur effective** du signal. Un signal subit un **évènement** si sa valeur effective au temps courant de simulation diffère de celle du temps précédent.

Avant la simulation, la phase de compilation se divise en deux étapes :

**La compilation frontale :** elle consiste en la vérification de la syntaxe et de la sémantique et traduit la description sous forme d'un arbre syntaxique. Le compilateur LVS de LEDA est un compilateur frontal.

**L'élaboration :** elle permet la réalisation d'un modèle simulable. L'élaboration consiste à créer les objets nommés de la description, ces objets peuvent porter un type (signaux, variables, constantes), et l'élaboration consiste pour eux à réserver de la place en mémoire. Ils peuvent aussi porter une valeur (expressions globalement statiques, c'est-à-dire calculables au temps 0), et l'élaboration consiste pour eux à mettre la valeur dans l'espace mémoire préalablement réservé. Lorsque ces objets portent une information liant entre eux des objets de différentes unités, l'élaboration va alors faire réellement ce lien: c'est la configuration.
L'élaboration effectue également l'instanciation: cette phase consiste en une "mise à plat" du système où chacune des instructions concurrentes est réécrite en process. En ce qui concerne les composants, ils sont remplacés par leur corps d'architecture en faisant l'instanciation `ports formels\ports effectifs`. Ainsi, toute une hiérarchie de composants est remplacée par le corps de leur architecture jusqu'à arriver à une description comportementale à base de process.

La plupart des simulateurs fonctionnent de la façon suivante : la description VHDL est traduite (en C le plus souvent) pour sa partie comportementale. Le code C sert simplement

d'assembleur portable. Les informations structurelles et "flot de données" sont traitées différemment selon les cas, mais, à la fin, on fait une édition des liens entre la partie système (le noyau fourni par le fabricant) et ce code C qui dépend de la description. Si aucune erreur n'est trouvée, on passe à l'exécution du modèle de simulation.

Le processus ("process") est ainsi l'objet fondamental manipulé par le simulateur. Une description VHDL se résume pour lui à un fouillis de process ayant chacun pour caractéristiques les signaux auxquels il est *sensible* et les opérations *séquentielles* qu'il exécute.
La norme VHDL déclare :

- Un process ne contient que des instructions séquentielles.
- Toute instruction concurrente peut toujours être traduite par un process (c'est le "process équivalent").
- Un process "vit" toujours, sa durée de vie est celle de la simulation. Il peut être **actif** ou **inactif** (interrompu temporairement ou définitivement sur un wait). Arrivé sur son mot clé final (**end process**), il s'exécute à nouveau depuis son mot clé de début **begin**. Un process est cyclique (itératif).

Un processus supplémentaire est construit : le **noyau**. Il a accès à l'ensemble des signaux et coordonne l'activité des processus définis par l'utilisateur. Lors de la simulation, il propage la valeur des signaux, détecte les évenements qui se produisent et réveille les processus adéquats en réponse à ces évènements.

L'élaboration achevée, nous avons un modèle simulable. La *simulation* consiste en l'exécution du modèle obtenu à l'issue de l'élaboration pendant un temps fixé par l'utilisateur. Elle permet le calcul et la visualisation de la valeur des signaux et des variables pendant cette durée. La simulation commence par une phase dite "**phase d'initialisation**" suivie d'une **exécution répétée** du cycle de simulation jusqu'au temps maximum donné par l'utilisateur (`time'high`). La simulation est arrêtée si ce temps est atteint. On présente ci-après les actions se produisant dans chacune des phases.

- Pendant l'initialisation :
  1. Le temps de simulation est mis à 0.
  2. Les signaux et les variables sont initialisés à leur valeur par défaut.
  3. Chaque processus est lancé et s'exécute jusqu'à sa suspension sur une instruction "wait" (implicite ou explicite).
  4. Le prochain temps de simulation est calculé.
- Un cycle de simulation comprend les actions suivantes :
  1. Mise à jour du temps de simulation. Si ce temps vaut `time'high`, la simulation s'arrête.

2. Mise à jour des signaux et détermination des évènements.
3. Réactivation des processus en attente sur un évènement qui vient de se produire.

Pour transformer une description VHDL en un modèle exécutable dans Acl2, nous devons formaliser le processus d'élaboration [9] et modéliser le cycle de simulation [10]. Nous ne nous occupons pas, dans ce chapitre, de la phase de compilation : nous supposerons à ce niveau que les descriptions ont franchi cette phase et qu'elles sont exemptes d'erreurs.

## 3.4 Le modèle sémantique

### 3.4.1 La phase d'élaboration

Chaque objet nommé de la description (notamment lors de déclarations) doit avoir un emplacement mémoire. Nous définissons alors un *état mémoire* comme étant une liste contenant toutes les valeurs des éléments mémorisants de la description (signaux, variables). Cet état peut être vu comme une *photographie* de la mémoire et des signaux d'interface.
Ainsi, pour un couple entité-architecture *entity_arch*, sont définies les fonctions suivantes :

**définition de l'état initial :** *entity_arch*_make_state prend en argument des valeurs initiales des signaux d'entrée et des signaux locaux. Cette fonction retourne l'état mémoire initial complet (comportant les valeurs par défaut).

**position des éléments de l'état mémoire :** *entity_arch*-get-nth prend en argument le nom d'un signal ou d'une variable et retourne sa position dans l'état mémoire.

**définitions des accesseurs de l'état :** *entity_arch*-getst permet d'accéder à la valeur d'un signal ou d'une variable. *entity_arch*-putst permet de modifier la valeur d'un signal ou d'une variable.

La notion de signal VHDL est modélisée par deux valeurs : la **valeur courante** et la **valeur future**. Ainsi, les signaux locaux et les signaux de sorties (**out**) sont modélisés par deux éléments. Si **Sig** est un signal déclaré, l'état mémoire contiendra la valeur courante étiquetée par **Sig** et la valeur future étiquetée par le nom **Sig+**. Une **variable** VHDL ne possède pas de pilote, et sa modification affecte sa valeur actuelle; elle est alors modélisée dans notre état mémoire par un seul élément étiqueté par le nom de la variable.

### 3.4.2 Les types de données

Comme mentionnés dans la section précédente, les types de données acceptés par notre sous-ensemble sont les suivants (les prédicats reconnaisseurs que nous avons définis sont indiqués) :

- Les entiers (integerp) et naturels (naturalp)
- Les bits (bitp)
- Les vecteurs de bits (array-bitp)

– Les types *signed* et *unsigned* (signedp et unsignedp)

Nous allons détailler chacun d'entre eux afin d'extraire leur sémantique :

**Les entiers :** ils représentent la même sémantique que la notion d'entiers formalisés dans Acl2 (cf chapitre précédent).

**Les bits :** sont modélisés par les naturels 0 et 1.

**Les vecteurs de bits :** sont modélisés par une liste de longueur fixée contenant les éléments naturels 0 et 1. Nous avons défini les accesseurs des éléments d'un vecteur :

**getarrayi :** est la fonction qui prend en argument le vecteur et un naturel $i$ avec $i <$ (longueur du vecteur) et qui retourne le $i^{eme}$ élément du vecteur (à partir de 0).

**setarrayi :** est la fonction qui prend un entier naturel $i$, un nouvel élément *new_el* et un vecteur et qui retourne le nouveau vecteur où le $i^{eme}$ élément a été changé par *new_el*.

**Les types signed et unsigned :** sont caractérisés par des entiers pour les signed et des entiers naturels pour les unsigned. Acl2 traite donc ce type de donnée par rapport à leur signification (entiers) plutôt qu'à leur représentation (vecteurs de bits). Ce type de données est issu de la bibliothèque **ihs** (Integer Hardware Specification) d'Acl2. Cette bibliothèque, développée par Bishop Brock pour la preuve du processeur DSP Motorola CAP [19], représente donc les bits et vecteurs de bits comme des entiers, accélérant ainsi la simulation en évitant de manipuler des listes conventionnelles de booléens. Les entiers 1 et 0 représentent un bit. Un vecteur $n$-bit est représenté par un entier dont la représentation binaire possède les mêmes bits significatifs.

Par exemple, un vecteur de 4 bits "1101" peut être représenté par l'entier 13. La bibliothèque IHS ne permet pas de fournir une méthode pour spécifier la longueur du vecteur de bits représentée par l'entier. Ainsi, 13 peut représenter le vecteur 1101 aussi bien que le vecteur "0000000000001101".

Les opérateurs logiques sur les bits sont définis dans cette bibliothèque :
`b-not, b-and, b-xor....`
Les opérateurs sur les vecteurs logiques sont définis séparément.

**lognot :** prend un entier représentant un vecteur de bit et retourne l'entier représentant son complément à 1.

**logand :** retourne le "et" logique de ses arguments. Par exemple :

```
(lognot 0) = -1
(lognot 1) = -2
(logand 3 5) = 1
(logand 3 -1) = 3
```

**logbit :** est la fonction qui prend en argument un entier naturel $i$ et un entier E, et qui rend le $i^{eme}$ bit logique de l'équivalent en vecteur de bits de E.

La figure 3.1 représente les fonctions IHS, utilisant le bit de poid le plus faible à gauche (début de liste).

### 3.4.3 Affectations de signaux et de variables

Il s'agit ici de modéliser les modifications de valeurs de l'état mémoire. Lorsque nous avons une affectation de variable $A := B$, il s'agit ici de prendre la valeur de B — par (*entity_arch*-getst B st) si *st* représente l'état mémoire — et de la placer à l'emplacement de A au moyen de (*entity_arch*-putst, nous avons donc :

st' = (*entity_arch*-putst A (*entity_arch*-getst B st) st)

Pour les affectations de signaux, le principe est le même, à la différence que la nouvelle valeur est mise à l'emplacement dédié à A+, la valeur future de A.

### 3.4.4 Instructions séquentielles

#### Les blocs d'instructions séquentielles

Chaque instruction entraîne la création d'un nouvel état mémoire, la fonction `desc-putst` rendant ce nouvel état. Les blocs d'instructions séquentielles sont modélisés par une macro **seq** qui engendre une succession d'appels à la fonction *let*.

Par exemple, supposons le bloc d'instructions séquentielles (A, B et C étant des variables de type entier):

```
A := B;
C := 5;
if A=C then
   B:=2;
       else
   B:=1;
end if;
```

Ce bloc est modélisé par :

```
(let ((st (entity_arch-putst A (entity_arch-getst B st))))
    (let ((st (entity_arch-putst C 5 st)))
        (let ((st (if (= (entity_arch-getst A st)
                         (entity_arch-getst C st))
                      (let ((st (entity_arch-putst B 2 st)))
                          st)
                      (let ((st (entity_arch-putst B 1 st)))
                          st))))
            st)))
```

| IHS Functions | Informal Description using the C language |
|---|---|
| `(bitp b)` | T if `b` is a bit. |
| `(bfix x)` | Coerce `x` to a bit. |
| `(zbp b)` | Bit-boolean converter. Nil if b is 1. Otherwise, T. |
| `(b1p b)` | Bit-boolean converter. T if b is 1. Otherwise, nil. |
| `(b-if b x y)` | If `b` is 1, return `x`. Otherwise, `y`. |
| `(b-not a)` | Bit negation. |
| `(b-and a b)` | Bit AND. |
| `(b-ior a b)` | Bit inclusive OR. |
| `(b-xor a b)` | Bit exclusive OR. |
| `(b-eqv a b)` | Bit equivalence. |
| `(b-nand a b)` | Bit NAND. `(b-not (b-and a b))` |
| `(b-nor a b)` | Bit NOR. `(b-not (b-ior a b))` |
| `(b-andc1 a b)` | `(b-and (b-not a) b)` |
| `(b-andc2 a b)` | `(b-and a (b-not b))` |
| `(b-orc1 a b)` | `(b-ior (b-not a) b)` |
| `(b-orc2 a b)` | `(b-ior a (b-not b))` |
| `(unsigned-byte-p n v)` | $0 \leq$ `v` $< 2^{\texttt{n}}$ |
| `(logbit n v)` | The `n`'th bit of bit-vector `v`. `(v >> n) & 0x1` |
| `(logand u v)` | Bitwise AND. `(u & v)` |
| `(logcar v)` | The least significant bit of `v`. `(v & 0x1)` |
| `(logcdr v)` | Bit vector `v` without the least significant bit. `(v >> 1)` |
| `(logcons b v)` | Concatenation of bit `b` to vector `v`. `(v << 1) \| b` |
| `(logior u v)` | Bitwise inclusive OR. `(u \| v)` |
| `(lognot v)` | 1's complement. `(~v)` |
| `(logxor u v)` | Bitwise exclusive OR. `(u ^ v)`. |
| `(loghead n v)` | The least significant `n` bits in bit vector `v`. `(v &` $2^{\texttt{n}} - 1$ `)` |
| `(logtail n v)` | Bit vector `v` without the least significant `n` bits. `(v >> n)` |
| `(logextu n m v)` | Sign-extend `m`-bit vector `v` to `n` bits. |
| `(logapp n u v)` | Concatenation of bit vectors. `u \| (v << n)` |
| `(rdb (cons n i) v)` | `n` bits of `v` from the `i`'th bit. `(v >> i) &` $2^{\texttt{n}} - 1$ |

.

FIG. 3.1 – *Listes des fonctions IHS. Nous utilisons des expressions C comme descriptions informelles.*

Afin d'améliorer la compréhension et la facilité d'écriture, nous avons écrit la macro *seq*: l'ensemble d'instructions ci-dessus se réécrit :

```
(seq st
     (entity_arch-putst A (entity_arch-getst B st))
     (entity_arch-putst C 5 st)
     (if (= (entity_arch-getst A st) (entity_arch-getst C st))
        (seq st
             (entity_arch-putst B 2 st))
        (seq st
             (entity_arch-putst B 2 st))))
```

Le schéma est donc le suivant :

```
<bloc d'instructions séquentielles> ::=
(seq st
     instr_1
     instr_2
     ...
     instr_n)
```

### Les boucles *for*

Les instructions *for...loop* sont utilisées pour répéter une séquence d'instructions, par exemple :

```
for I in 1 to 50 loop
    M(i) := M(i+1);
end loop;
```

Il s'agit de modéliser l'instruction *for* en fonction récursive. Il est donc nécessaire d'exhiber une "mesure" qui décroît et une condition d'arrêt. Nous n'acceptons ici que les instructions *for* contenant une borne de départ (ici 1) et une borne d'arrivée (ici 50).

L'instruction *for...loop* est remplacée par un appel de fonction. La définition de cette fonction dépend de la direction (ascendante ou descendante) de la boucle. L'instruction ci-dessus (que nous nommons **for1**) est remplacée alors par l'appel `(for1 1 50 st)`.

Pour les boucles de la forme `for I in a to b`, où `a<b`, la fonction récursive est calquée sur le schéma ci-dessous. Précisons que `a` et `b` sont les instances de i et j, la fonction étant ensuite invoquée par `(name_for a b st)`.

```
(defun name_for (i j st)
       (declare (xargs :measure (nfix (- j i))))
       (if (zp (- j i))
          st
          (name_for (1+ i)
                    j
                    <bloc d'instructions séquentielles>)))
```

La partie `(declare (xargs :measure (nfix`[1] `(- j i))))` spécifie la mesure (non triviale dans ce cas) décroissante pour la récursion.

### 3.4.5 Les process, l'architecture, le cycle de simulation

Nous modélisons la notion de process par une fonction prenant en argument l'état mémoire. La fonction contient les blocs d'instructions séquentielles. Voici le schéma adopté pour la modélisation des process :

```
(defun nom_du_process (st)
        <bloc d'instructions séquentielles>)
```

L'*architecture* décrit le comportement d'une description VHDL, la fonction qui la modélise doit appeler les process qui la composent. Cette fonction modélise ainsi un *cycle de simulation* sans mise à jour des pilotes.
L'ordre des appels des process est arbitraire mais n'a pas de conséquence sur l'état final obtenu. En effet, un process ne modifie qu'une partie distincte de l'ensemble des signaux (ou variables). Voici le schéma de la fonction modélisant l'architecture :

```
(defun entity_arch-cycle (st)
        (seq st
             (nom_du_process_1 st)
             (nom_du_process_2 st)
...
             (nom_du_process_n st)))
```

La *mise à jour* des pilotes doit intervenir après exécution de l'instruction **wait** par tous les processus actifs en VHDL. Elle consiste à mettre à jour la valeur actuelle d'un signal **Sig** par la valeur future **Sig+**.

Supposons que l'ensemble $x_1, x_2, ..., x_n$ représente l'ensemble des signaux de sortie ou locaux à l'architecture. La fonction modélisant la mise à jour des pilotes se résume au schéma suivant :

```
(defun entity_arch-update-signals (st)
        (seq st
             (entity_arch-putst x_1 (entity_arch-getst x_1+ st) st)
             (entity_arch-putst x_2 (entity_arch-getst x_2+ st) st)
             ...
             (entity_arch-putst x_n (entity_arch-getst x_n+ st) st)))
```

Le mécanisme des cycles de simulation est ainsi représenté par une fonction récursive prenant en argument l'état mémoire et le nombre souhaité de cycles de simulation. Cette fonction

1. `Nfix retourne l'argument si celui-ci est un naturel, 0 sinon.`

applique la mise à jour des pilotes avant de faire tourner les process. Voici le schéma de la fonction :

```
(defun entity_arch-simul (n st)
    (if (zp n)
        st
        (entity_arch-simul (1- n)
                           (entity_arch-cycle
                                        entity_arch-update-signals st))))
```

### 3.4.6 Exemple : la factorielle de n

Pour des raisons de lisibilité, les fonctions Acl2 de cet exemple s'appelent *fact*-xxx et non *mycomputation-fact*-xxx

Prenons comme exemple une description VHDL réalisant un calcul de factorielle (figure 3.2). Il s'agit d'une description VHDL contenant deux process : "Mult" et "Doit". Le premier process est la partie opérative de la description, réalisant une multiplication. Le second est la partie contrôle. Ainsi, la description réalise, par multiplications successives, une factorielle de l'entrée *arg* si le signal d'entrée *start* est à vrai. Le résultat est exprimé sur la sortie *output*.

La formalisation complète en Acl2 est donnée en annexe. L'état mémoire (où l'horloge a été supprimée) :

```
(arg start ;valeurs courantes des signaux d'entrées
 op1 op2 resmult startmult endmult ;valeurs courantes des signaux locaux
 op1+ op2+ resmult+ startmult+ endmult+ ;valeurs futures des signaux locaux
 mystate r f                          ;variables déclarées dans le process Doit
 res done                             ;valeurs courantes des signaux de sortie
 res+ done+)                          ;valeurs futures des signaux de sortie
```

possède comme valeur initiale :

```
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

Le typage des éléments déclarés est modélisé sous forme d'une fonction qui va prendre le rôle de prédicat reconnaisseur du typage de l'état mémoire, nous la nommons *entity-arch-stp*. Cette fonction vérifie donc qu'après modification, l'état mémoire est toujours bien typé, c'est-à-dire que les éléments (signaux ou variables) de l'état-mémoire conservent leur type de données. Par exemple, pour la description fact, le prédicat `fact-stp` est défini comme suit:

```
(defun fact-stp (st)
        (and (true-listp st)
        (= (length st) 19)
        (naturalp (fact-getst 'arg st))
        (bitp (fact-getst 'start st))
        (naturalp (fact-getst 'op1 st))
        (naturalp (fact-getst 'op2 st))
        (naturalp (fact-getst 'resmult st))
```

```
entity mycomputation is
       port (arg : in natural;
             start,clk : in bit;
             output : out natural;
             done : out bit);
end mycomputation;
architecture fact of mycomputation is
begin
      signal op1,op2,resmult : natural;
      signal startmult,endmult : bit;
      Mult: process
           begin
                wait UNTIL clk='1';
                if startmult='1' then
                  resmult <= op1*op2;
                end if;
                endmult <= startmult;
      end process Mult;
      Doit: process
           variable mystate : natural := 0;
           variable R,F : natural := 0;
      begin
           wait UNTIL clk='1';
           case mystate is
                when 0 => R := arg;
                          F := 1;
                          if start='1' then
                            mystate := '1';
                          end if;
                when 1 => if R = '1' then
                            output <= F;
                            done <= '1';
                            mystate := 0;
                          else
                            startmult <= '1';
                            op1 <= R;
                            op2 <= F;
                            mystate := 2;
                          end if;
                when 2 => if endmult = '1' then
                            F := resmult;
                            R := R-1;
                            startmult <= '0';
                            mystate := 1;
                          end if;
           end case;
      end process Doit;
end fact;
```

FIG. 3.2 – *description VHDL d'un calcul de factorielle*

```
(bitp (fact-getst 'startmult st))
(bitp (fact-getst 'endmult st))
(naturalp (fact-getst 'op1+ st))
(naturalp (fact-getst 'op2+ st))
(naturalp (fact-getst 'resmult+ st))
(bitp (fact-getst 'startmult+ st))
(bitp (fact-getst 'endmult+ st))
(naturalp (fact-getst 'mystate st))
(naturalp (fact-getst 'r st))
(naturalp (fact-getst 'f st))
(naturalp (fact-getst 'res st))
(bitp (fact-getst 'done st))
(naturalp (fact-getst 'res+ st))
(bitp (fact-getst 'done+ st))))
```

### 3.4.7 Hiérarchie de composants

La technique de modélisation des process a besoin d'être étendue afin de pouvoir traiter les architectures avec composants. Reprenons comme exemple notre description de la factorielle 3.2 que nous utilisons comme composant d'une nouvelle description représentée sur la figure 3.3.

*L'état mémoire de l'architecture la plus englobante contient les états mémoires des composants.* Par exemple, l'état mémoire de l'architecture *factorial-comp*, modélisé sous forme d'une liste, va contenir un élément qui est la liste représentant l'état mémoire du composant FACT_UNIT instancié. L'état mémoire de architecture factorial-comp se représente ainsi :
(`E START S FACT_UNIT S+ DONE DONE+`), où `FACT_UNIT` est un état mémoire pour l'architecture **mycomputation** de **fact**, c'est-à-dire la liste

```
(arg start ;valeurs courantes des signaux d'entrées
 op1 op2 resmult startmult endmult ;valeurs courantes des signaux locaux
 op1+ op2+ resmult+ startmult+ endmult+ ;valeurs futures des signaux locaux
 mystate r f                            ;variables déclarées dans le process Doit
 res done                               ;valeurs courantes des signaux de sortie
 res+ done+)                            ;valeurs futures des signaux de sortie
```

Comme les process, les composants sont modélisés par une fonction de transition. Cette fonction transfère les valeurs des ports effectifs vers ou depuis l'état mémoire du composant, et effectue un cycle de simulation du composant en appelant la fonction de transition liée à son l'architecture.

Les encadrés ci-dessous illustrent les compléments du modèle Acl2 pour les architectures utilisants des composants. Nous notons

- *entity-arch* le couple entité/architecture modélisant l'architecture la plus englobante, afin de caractériser les fonctions *entity-arch*-getst, *entity-arch*-putst, *entity-arch*-update-signals, *entity-arch*-cycle, etc.

```
entity comp is
        port (E : in natural;
              START,CLK : in bit;
              S : out natural;
              DONE : out bit);
end comp;

architecture factorial-comp of comp is
begin
      component FACT_UNIT
          port (arg : in natural;
                start,clk : in bit;
                output : out natural;
                done : out bit);
      end component;
begin
– instanciation du composant FACT_UNIT
      C: FACT_UNIT
         port map (arg => E,
                   start => START,
                   clk => CLK,
                   output => S,
                   done => DONE);
end factorial-comp;
configuration config_fact of comp is
for C : FACT_UNIT
    use entity mycomputation(fact);
end for
end config_fact;
```

FIG. 3.3 – *Architecture contenant comme composant la description 3.2*

- $c_i$ le nom d'un composant i (couple entité-architecture) utilisé dans l'architecture. Nous notons *comp-*$c_i$-getst, *comp-*$c_i$-putst, *comp-*$c_i$-cycle, *comp-*$c_i$-update-signals, les fonctions définies lors de la formalisation du couple entité/architecture représentant le composant. Il est à noter que si le composant est instancié plusieurs fois, les mêmes fonctions ci-dessus sont utilisées sur des états-mémoires différents.
- $a_i$ étant des signaux locaux à l'architecture la plus englobante, ou des signaux d'entrée. Ces signaux ont pour vocation d'être branchés aux entrées des composants.
- $b_i$ étant des signaux locaux à l'architecture la plus englobante, ou des signaux de sortie. Ces signaux ont pour vocation d'être branchés aux sorties des composants.
- $e_i$-$c_i$ est le signal d'entrée $e_i$ du composant $c_i$.
- $s_i$-$c_i$ est le signal de sortie $s_i$ du composant $c_i$.

La fonction de mise à jour des signaux de l'architecture **factorial-comp** est étendue avec plusieurs instructions supplémentaires propres au composant. Il s'agit des raccordements entre ports formels du composants et signaux de l'architecture englobante.

```
(defun entity-arch-UPDATE-SIGNALS (st)
(seq st
...
;;; Mise à jour des pilotes de l'architecture et des composants
        (entity-arch-putst
        'c_i
        (comp-c_i-update-signals (entity-arch-getst
                                  'c_i st)) st)
...
;;; Affectation des ports d'entrées des composants
        (entity-arch-putst 'c_i
                           (comp-c_i-putst 'e_i-c_i
                                           (entity-arch-getst 'a_i st)
                                           (entity-arch-getst 'c_i st)) st)
...
;;; Transmission des valeurs des sorties des composants vers l'architecture
        (entity-arch-putst 'b_i+
                           (comp-c_i-getst 's_i-c_i+
                                           (entity-arch-getst 'c_i st)) st)))
```

La fonction *entity-arch*-cycle de l'architecture englobante est également étendue afin d'exécuter les fonctions $component_1$-cycle de chaque composant. Ainsi le schéma est le suivant :

```
(defun entity-arch-cycle (st)
        (seq st
             ...
;; Complément pour l'utilisation des composants
             (entity-arch-putst 'c_i
                                (comp-c_i-cycle (entity-arch-getst 'c_i st)))))
```

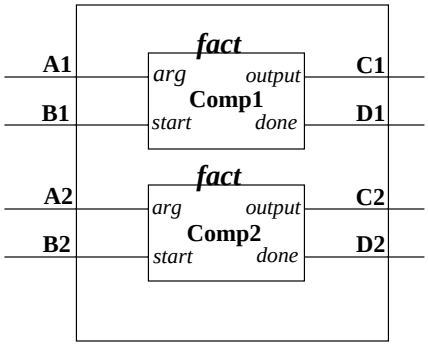


FIG. 3.4 – *Architecture regroupant deux composants réalisant chacun une factorielle*

Prenons par exemple une architecture contenant 2 composants "fact", comme l'illustre la figure . La fonction de mise à jour des pilotes de l'architecture englobante est étendue suivant le schéma suivant :

```
(defun entity-arch-UPDATE-SIGNALS (st)
... ;;; Mise à jour des pilotes de l'architecture englobante
;;; Mise à jour des pilotes des composants
        (entity-arch-putst
        'Comp1
        (fact-update-signals (entity-arch-getst
                              'Comp1 st)) st)
        (entity-arch-putst
        'Comp2
        (fact-update-signals (entity-arch-getst
                              'Comp2 st)) st)
...
;;; Affectation des ports d'entrées des composants
        (entity-arch-putst 'Comp1
                           (fact-putst 'arg
                                       (entity-arch-getst 'A1 st)
                                       (entity-arch-getst 'Comp1 st)) st)
        (entity-arch-putst 'Comp1
                           (fact-putst 'arg
                                       (entity-arch-getst 'B1 st)
                                       (entity-arch-getst 'Comp1 st)) st)
        (entity-arch-putst 'Comp2
                           (fact-putst 'arg
                                       (entity-arch-getst 'A2 st)
                                       (entity-arch-getst 'Comp2 st)) st)
        (entity-arch-putst 'Comp2
                           (fact-putst 'arg
                                       (entity-arch-getst 'B2 st)
                                       (entity-arch-getst 'Comp2 st)) st)
...
;;; Transmission des valeurs des sorties du composant vers l'architecture
        (entity-arch-putst 'C1+
```

```
                (fact-getst 'output+
                        (entity-arch-getst 'Comp1 st)) st)
(entity-arch-putst 'D1+
                (fact-getst 'done+
                        (entity-arch-getst 'Comp1 st)) st)
(entity-arch-putst 'C2+
                (fact-getst 'output+
                        (entity-arch-getst 'Comp2 st)) st)
(entity-arch-putst 'D2+
                (fact-getst 'done+
                        (entity-arch-getst 'Comp2 st)) st))
```

Les modèles des architectures avec composants peuvent être utilisés lors d'exécution, simulation symbolique et vérification formelle de la même façon qu'une architecture ordinaire. Cependant, il est préférable de diviser l'effort de preuve au travers de la hiérarchie des composants. Les propriétés prouvées sur un composant doivent pouvoir être utilisées pour toutes les descriptions l'utilisant.

### 3.4.8 Les fonctions VHDL

*Les valeurs des paramètres peuvent changer entre les différents appels et donnent ainsi des exécutions différentes.* La définition d'une fonction est sémantiquement similaire à la notion de composant dans le sens où nous pouvons considérer une fonction VHDL comme une boîte noire, dans laquelle nous pouvons voir les entrées (arguments) et une sortie (le résultat). Nous sommes ainsi proches de la notion d'entité VHDL. Le corps de la fonction peut être vu comme un process combinatoire qui initialiserait toutes ses variables locales à chaque cycle d'exécution. Ainsi, même si la sémantique de simulation n'est pas définie de la même façon que dans le standard IEEE, les appels de fonctions peuvent être modélisés comme des instanciations d'un composant combinatoire.
Ainsi, un état mémoire et ses accesseurs sont définis. Une variable nommée `return` caractérise la "sortie" de la fonction.

Considérons la fonction VHDL suivante :

```
function div4(d : std_logic_vector) return std_logic_vector is
        variable r : std_logic_vector(8 downto 0);
begin
    if (d(16) = '1') then
      r := (8 => '1', others => '0');
    else
      r := (8 => '0', others => '1');
    end if;
    return r;
end;
```

Nous créons un état mémoire et ses accesseurs:

- div4-get-nth
- div4-putst
- div4-getst
- div4-stp

Les éléments de l'état mémoire sont les suivants :
*(d r return)*

La fonction Acl2 `div4` se modélise ainsi : (*others l i j k .....* est un constructeur d'un vecteur de bits de taille l avec les $i^{th}$, $j^{th}$, $k^{th}$, ... bits à '1'.)

```
(defun DIV4 (d)
  (let ((st (div4-putst 'd (list d) (div4-make-state))))
     (seq st
         (if (= (getarrayi (div4-getst 'd st) 16) 1)
           (seq st (div4-putst 'r (others 9 8) st))
           (seq st (div4-putst 'r (others 9 0 1 2 3 4 5 6 7) st)))
         (div4-putst 'return 'r st))
     (div4-getst 'return st)))
```

Ainsi, l'utilisation de la fonction VHDL div4 se modélise par un appel de la fonction en Lisp. Considérons un couple entité-architecture `fmt-rtl`, l'appel sera :

```
(div4 (fmt-rtl-getst 'd st))
```

### 3.4.9 Simulation numérique

#### Simulation et analyse formelle

La simulation d'un système fait partie de son développement, car tester un système pendant la phase de conception est très utile: elle permet de prédire les performances du système; de détecter des erreurs et d'aider la conception mixte matérielle-logicielle. Les modèles simulables ont également été construits pour une analyse formelle du système. Le mariage entre la simulation et l'analyse formelle pendant la phase de conception est avantageux. Il permet, grâce à un modèle formel unique, non seulement de pouvoir le "faire tourner", mais également de travailler sur des propriétés du système, renforçant ainsi la confiance que peut avoir le concepteur sur ce modèle.

Construire un modèle dans Acl2 permet de disposer d'un modèle exécutable dans un environnement formel. Ce chapitre présente la phase d'exécution du modèle Acl2. Il présente les optimisations faites sur le modèle pour une simulation efficace.

### Simulation du modèle

Prenons l'architecture **fact**, que nous avons vu précédemment (figure 3.2),
Nous prenons comme définition des accesseurs *arch*-putst et *arch*-getst les suivantes :

(**defun** *entity-arch*-getst (var st)
(nth (*entity-arch*-get-nth var) st))

(**defun** *entity-arch*-putst (var new st)
(update-nth (*entity-arch*-get-nth var) new st))

L'état initial doit être fourni par l'utilisateur comme une liste explicite de valeurs, comme suit, le résultat de la factorielle de 7 étant 5040 et la valeur de done = 1.

```
ACL2 !>(fact-simul 19 '(7 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
(7 1 2 2520 5040 1 1 2 2520 5040 0 1 1 1 5040 0 0 0 0)
ACL2 !>(fact-simul 20 '(7 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
(7 1 2 2520 5040 0 1 2 2520 5040 0 0 0 1 5040 0 0 5040 1)
```

Cependant, pour les circuits disposant d'états-mémoire de plusieurs dizaines ou centaines d'éléments, il n'est pas commode d'écrire de telles listes.

**La macro *entity-arch*-make-state:**

- L'initialisation peut se faire grâce à cette macro, que nous avons définie, qui affecte la valeur par défaut du signal ou de la variable par rapport à son type. Pour un couple entité-architecture donné, cette fonction s'appele *entity-arch*-make-state. (*entity-arch*-make-state) retourne l'état-mémoire par défaut.

- Les valeurs arbitraires sont données par l'utilisateur au moyen des arguments de la fonction *entity-arch*-make-state. Par exemple (*entity-arch*-make-state :A '5 :B '4) retourne un état-mémoire avec les valeurs par défaut sauf pour les éléments A et B où les valeurs '5 et '4 sont affectées.

Par exemple, pour l'architecture **fact**, la fonction (qui est en fait construite sous la forme d'une macro Common Lisp) possède la définition suivante :

```
(defmacro FACT-MAKE-STATE
(&key (arg '0)
      (start '0)
      (op1 '0)
      (op2 '0)
      (resmult '0)
      (startmult '0)
      (endmult '0)
      (op1+ '0)
      (op2+ '0)
```

```
      (resmult+ '0)
      (startmult+ '0)
      (endmult+ '0)
      (mystate '0)
      (r '0)
      (f '0)
      (res '0)
      (done '0)
      (res+ '0)
      (done+ '0))
(list 'quote
(list arg
      start op1 op2 resmult startmult endmult
      op1+ op2+ resmult+ startmult+ endmult+
      mystate r f res done res+ done+))
)
```

Ainsi, pour le même exemple de simulation que ci-dessus, nous pouvons utiliser cette macro :

```
ACL2 !>(fact-simul 20 (fact-make-state :arg 7 :start 1))
(7 1 2 2520 5040 0 1 2 2520 5040 0 0 0 1 5040 0 0 5040 1)
```

## 3.5 Efficacité du modèle

Divers projets de vérification ont utilisé des démonstrateurs de théorèmes pour analyser des modèles de systèmes [19, 17, 87] transformant ainsi le démonstrateur en outil de conception [41]. L'exécution doit être rapide, similaire en vitesse aux modèles habituellement utilisés. De plus, le langage doit être capable de s'intégrer à des environnements de simulation et de débogage, pour être utilisable dans une chaîne de conception.

Acl2 possède la particularité suivante : chaque fois qu'une fonction *f* est définie dans Acl2, une fonction de même nom est définie dans le compilateur Common Lisp sur lequel est construit Acl2[2]. Cette version est appelée version "Raw Lisp" de *f*. Elle est utilisée lorsque des gardes[3] sont définies avec *f*. Par exemple, reprenons la définition de la fonction calculant une factorielle, en déclarant les gardes, elle devient :

```
(defun fact (n)
  (declare (xargs :guard (and (integerp n) (<= 0 n))))
   (if (zp n)
       1
      (* n
        (fact (- n 1)))))
```

2. Appelé "The underlying Common Lisp", il s'agit du compilateur Common Lisp servant de base à Acl2 : gcl, Allegro, MCL, ...

3. Déclarations du domaine des arguments d'une fonction

Cette nouvelle définition accepte seulement des entiers naturels comme arguments pour l'exécution. La compilateur attaché à Acl2 tire un avantage non négligeable pour accélérer la vitesse d'exécution.

De plus, certains compilateurs Lisp, en particulier GCL, peuvent grandement tirer partie des déclarations de type. En effet, une des sources d'inefficacité est l'arithmétique non déclarée. La plupart des langages représentent les entiers comme des mots machine, allouent la mémoire pour stocker arbitrairement de larges entiers et créent un pointeur "entiers" vers ces nombres. L'arithmétique d'Acl2 invoque ainsi des références mémoire et des vérifications de type qui peuvent rendre l'exécution très lente. Cela peut être évité en déclarant des domaines et, si cela est connu, en les limitant entre deux valeurs précises[86]. Par exemple, pour déclarer qu'une variable représente un entier entre -32768 et 32767, il faut déclarer x avec : *declare (type (integer -32768 32767))*. L'instruction *type* peut également accueillir un prédicat pour limiter un domaine, par exemple (type (satisfies true-listp) st) limite le domaine aux listes se terminant par nil. La déclaration *the* présente dans :

```
(defun f (x)
  (declare (type (integer -32768 32767) x))
 (the (integer -32766 32769) (+ x 2)))
```

déclare le domaine de sortie de la fonction f comme étant un entier compris entre -32766 et 32769 inclus.

Pour assurer une exécution rapide, les accesseurs *entity-arch*-getst et *entity-arch*-putst et la fonction *entity-arch*-get-nth sont définis avec les notions de gardes et d'optimisation de type:

```
;; Partie declare de la définition de l'accesseur Fact-getst

(defun FACT-GETST (var st)
      (declare (xargs :guard t)
               (type (member arg
                     start op1 op2 resmult startmult endmult
                     op1+ op2+ resmult+ startmult+ endmult
                     mystate r f res done res+ done+)
               var)
      (type (satisfies true-listp) st))
      (nth (FACT-GET-NTH var) st)) ;; corps de la fonction

;; Partie declare de la définition de l'accesseur Fact-putst

(defun FACT-PUTST (var new st)
      (declare (xargs :guard t)
               (type (member arg
                     start op1 op2 resmult startmult endmult
                     op1+ op2+ resmult+ startmult+ endmult+
                     mystate r f res done res+ done+)
               var)
               (type (satisfies true-listp) st))
```

```
        (update-nth (FACT-GET-NTH var) new st)) ;; corps de la fonction

;; Partie declare de la définition de la fonction fact-get-nth

(defun FACT-GET-NTH (var)
        (declare (type (member arg
                               start op1 op2 resmult startmult endmult
                               op1+ op2+ resmult+ startmult+ endmult+
                               mystate r f res done res+ done+)
                       var)
                 (xargs :guard t))
        (the (integer 0 18)

(case var ('arg 0)
          ('start 1)
          ('op1 2)
          ('op2 3)
          ...
          ('done+ 18)))
)
```

# Chapitre 4

# Simulation symbolique

## 4.1 Utilité de la simulation symbolique

La spécification formelle exécutable permet aux ingénieurs de tester (ou *simuler*) le système spécifié sur des données concrètes avant que le système soit implémenté. La spécification formelle d'une description dans un langage tel que Lisp sert le même but, mais peut encourager un style de spécification plus abstraite et surtout, fournit une voie pour la migration vers la preuve formelle.
*Parce que l'évaluation et la simulation sont déjà des techniques bien comprises des ingénieurs de conception, l'utilisation de spécifications formelles requiert peu ou pas de formation, malgré les notions logiques qui en font leur base.*

Cependant, cette idée est en contradiction avec l'esprit des méthodes formelles et des démonstrateurs de théorèmes qui tend à abstraire au maximum sur une logique. Une conciliation entre les deux idées serait de permettre que la spécification formelle soit utilisée pour faire de la *simulation symbolique*.

Le concept de simulation logique symbolique a été introduit par John Darringer au sein d'IBM en 1979 [32, 47]. Randal E. Bryant a été le premier a construire un simulateur symbolique au niveau bit en 1985 [20]. Depuis, des simulateurs symboliques ont été écrits pour des usages internes et bon nombre de vérifications ont été effectuées grâce à cette technique[23, 33, 8]. Avec un simulateur symbolique, un ingénieur peut "faire tourner" un modèle d'une description de circuit avec un ou plusieurs symboles en entrée. Ces symboles représentent l'ensemble des valeurs que peuvent accepter ces entrées. La simulation symbolique exhibe les sorties du modèle en fonction de ces symboles, montrant la fonctionnalité effectuée. Sa proche relation avec la simulation la rend facile à appréhender.

Avec ce type de technique, la spécification formelle peut être inspectée et analysée par les personnes connaissant la description et ses applications, de façon plus efficace et rapide.

## 4.2 Implémentation d'un simulateur symbolique

Jusqu'à présent, tout ce que nous avons obtenu est un cycle de simulation pour notre modèle. Pour abstraire l'expression obtenue en fonction de paramètres d'entrées, l'opération consiste à accumuler les expressions symboliques, calculées sur un ou plusieurs cycles de simulation, dans les variables de l'état-mémoire. Le but n'étant pas de remplacer toutes les valeurs des variables et signaux par des symboles, mais seulement celles qui peuvent être considérées comme des paramètres de la partie opérative du modèle, ou des signaux de contrôle qui dépendent de ces paramètres, tels qu'un signal de débordement ou le résultat booléen d'un test. Reprenons l'exemple de la description VHDL formalisée dans Acl2 représentant le calcul de la factorielle (figure 3.2). Rappelons le contenu de l'état-mémoire:

```
(arg start ;valeurs courantes des signaux d'entrées
 op1 op2 resmult startmult endmult ;valeurs courantes des signaux locaux
 op1+ op2+ resmult+ startmult+ endmult+ ;valeurs futures des signaux locaux
 mystate r f)                          ;variables déclarées dans le process Doit
 res done                              ;valeurs courantes des signaux de sorties
 res+ done+                            ;valeurs futures des signaux de sorties
```

Dans l'architecture `fact`, il est important que la valeur de la variable `mystate` soit initialement à 0. La valeur initiale est automatiquement fournie par `fact-make-state`. Inversement, nous sommes intéressés de garder la valeur de `arg` symbolique (nommons-la $Q$), et de vérifier que la variable `f` interne au process `DoIt`, ou le signal équivalent `resmult`, accumule bien les expressions de la forme $Q * (Q-1) * (Q-2)$...

Il existe deux méthodes permettant de réaliser une simulation symbolique, la première consiste à utiliser la commande *defthm*, la seconde utilise la bibliothèque "symsim", originellement développée par Matt Kaufmann [58].

### 4.2.1 Simulation symbolique par defthm

La simulation symbolique par la commande *defthm* est une solution inspirée de [61], dans laquelle Acl2 est utilisé pour produire des simplifications d'expressions symboliques, sur un modèle d'une micro-machine manipulée par quelques instructions assembleur. Pour l'adapter à notre modèle, nous devons considérer les mises à jour des pilotes ainsi que les appels aux process.

L'idée est la suivante : Lorsque Acl2 essaie de prouver un théorème, par exemple
`(equal (f x) (g x))`,
l'une des premières opérations appliquées par Acl2 est la simplification de termes (cf figure 2.2). Cette simplification consiste à réécrire les termes en utilisant la base de règles de réécriture existantes. L'idée de Moore dans [61] est d'essayer de faire prouver un théorème de la forme
`(equal (f x) v)`,
où v n'est pas spécifié. Ce théorème ne peut être prouvé mais Acl2 va appliquer ses heuristiques de simplification sur (f x) pour "essayer" de réduire (f x) à v. Cette transformation va "déplier" automatiquement toutes les fonctions par leurs définitions (sauf pour les fonctions récursives où nous devons ajouter de nouvelles règles de réécritures).

### Simulation symbolique de la factorielle

Ainsi, en premier lieu, pour effectuer une simulation symbolique de notre factorielle, il nous faut ajouter deux nouvelles règles de réécriture sur la fonction récursive `fact-simul`:

- La première règle doit ajouter l'information que simuler 0 fois ne change pas l'état, donc que (`fact-simul` 0 memory-state) = memory-state.
- il nous faut ensuite décomposer l'appel (`fact-simul` n st)
  en
  (`fact-simul` (1- n) (fact-cycle (fact-update-signals st))).

Ainsi, le théorème à fournir est :

```
(defthm unfold-fact-simul
  (and (equal (fact-simul 0 st) st)
       (implies (and (integerp n) (> n 0))
                (equal (fact-simul n st)
                       (fact-simul (1- n)
                                   (fact-cycle
                                        (fact-update-signals st))))))
  :rule-classes :rewrite)
```

Le théorème "faux" à soumettre à Acl2 est donné ci-dessous; `st` est l'état-mémoire initial avec la donnée d'entrée *arg*, associé à un entier naturel symbolique $q$. Le signal de contrôle *start* est mis à 1. Il est nécessaire de préciser un nombre fixé de cycles de simulation car nous ne pouvons avoir des expressions de longueur infinie (ici, nous fixons 12). Le théorème que nous essayons de prouver indique l'égalité entre (`fact-simul` 12 st—) et une variable inconnue `v`.

```
(defthm symbolic_simulation                                    _
  (implies (and (equal st (fact-make-state :arg q :start 1))    |
                (integerp q)                                    | Hypotheses
                (>= q 0))                                      _| du théorème

           (equal (fact-simul 12 st)
                  v))
  :otf-flg t
  :hints (("Goal" :do-not '(generalize
                            fertilize
                            eliminate-destructors
                            eliminate-irrelevance)
                  :in-theory (disable COMMUTATIVITY-OF-*)))
  :rule-classes nil)
```

Les "hints" correspondent à des guides pour le démonstrateur: ici, l'instruction `:do-not` demande de ne pas exécuter les heuristiques de preuve énumérées en argument (généralisation, fertilisation, ...).
`:in-theory (disable COMMUTATIVITY-OF-*)` demande au démonstrateur de ne pas employer

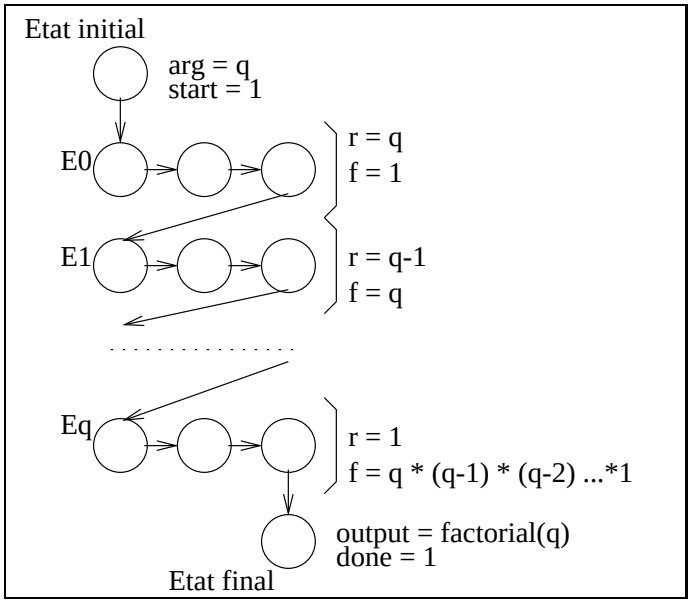


FIG. 4.1 – *Modélisation des cycles de calcul du modèle factoriel*

la règle de commutativité de l'opérateur de multiplication, afin de garder l'expression finale sous la forme `(* q (q -1) ..)`

`:rule-classes nil` signifie que l'on ne désire pas employer le théorème comme nouvelle règle de déduction.

La directive `:otf-flg` demande au simplificateur d'explorer et de simplifier tous les sous-buts.

La représentation des cycles de calcul est représentée sur la figure 4.1. Après un cycle de simulation à partir de l'état initial, la variable `mystate` prend la valeur 1. Nous sommes dans le cas où la partie contrôle débute son *cycle de calcul*. Nous pouvons observer que ce cycle de calcul représente 3 cycles de simulation (de E0 à E1):
- E1 et E0 possèdent une valeur de `mystate` égale à 1, on dit alors que `mystate` est un invariant du cycle de calcul, de la même manière, le signal `startmult` est à 0.
- La variable $r$ devient $r - 1$.
- La variable $f$ accumule le résultat du cycle précédent.

Nous observons ainsi une symétrie dans la représentation des cycles de calcul par rapport aux cycles d'horloge. Ainsi, pour réaliser un calcul de factorielle pour une valeur $q$, il est nécessaire de fournir $3 * q$ cycles de simulation après le cycle d'initialisation pour avoir le résultat sur la variable $f$.

Le théorème `symbolic_simulation` prend 12 cycles de simulation à partir de l'état initial, soit 1 cycle de simulation pour l'initialisation puis 11 cycles qui représentent 3 cycles entier de calcul et l'arrêt sur l'état E' de la figure 4.2. Cependant si $q$ est égal à 1, le système s'arrête,

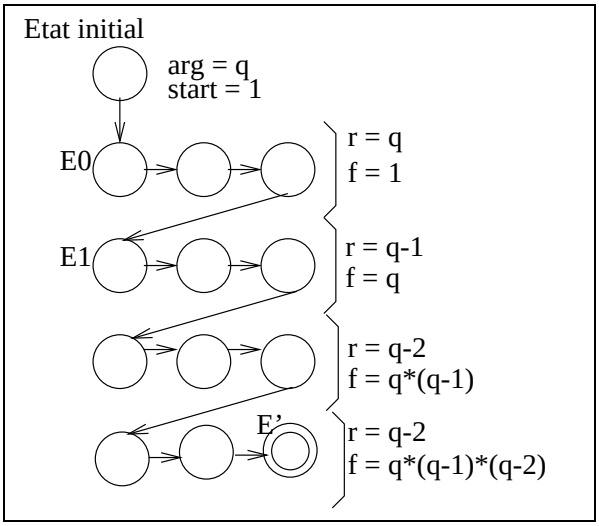


FIG. 4.2 – *Représentation en cycles de calcul de 12 cycles de simulation*

car dans le cas où `mystate`=1 et R=1, le calcul se termine, la sortie `output` prend la valeur de f, `done` passe à true. De même si $q = 2$, le système n'ira pas au bout des cycles de calcul car au prochain cycle d'horloge, q vaudra 1, le système s'arrêtera alors pour la même raison que ci-dessus. Lorsque $q = 3$, le système finit son calcul après ces 12 cycles de simulation. Le cas général est le cas où $q > 3$. Ainsi, Acl2 génère 4 sous-buts différents en fonction de la valeur initiale de $q$. Ces sous-buts sont explicités ci-dessous:

**Si** $q = 1$ **:** Les valeurs de r, f, res, done, res+, done+ prennent la valeur 1.

```
(implies (equal q 1)
         (equal '(1 1 0 0 0 0 0 0 0 0 0 0 0
                  1  <-- r
                  1  <-- f
                  1  <-- res
                  1  <-- done
                  1  <-- res+
                  1) <-- done+
                v))
```

**Si** $q = 2$ **:** Le système n'effectue qu'un cycle de calcul (pour q=2). Nous obtenons `(* q 1)` en résultat.

```
(implies (and (integerp q)
              (<= 0 q)
              (not (equal q 1))
              (equal (+ -1 q) 1))
         (equal (list q 1 q 1 (* q 1)
                      0 0 q 1 (* q 1)
```

```
                    1 0 2
                    q       <-- r
                    1       <-- f
                    (* q 1) <-- res
                    1       <-- done
                    (* q 1) <-- res+
                    1)      <-- done+
                v))
```

**Si** $q = 3$ **:** Le système effectue 2 cycles de calcul, pour $q = 3$, et $q = 2$, le résultat est donc `(* q (+ -1 q) 1)`.

```
(implies (and (integerp q)
              (<= 0 q)
              (not (equal q 1))
              (not (equal (+ -1 q) 1))
              (equal (+ -2 q) 1))
         (equal (list q 1 q 1 (* q 1)
                      1 1 q 1 (* q 1)
                      0 1 1
                      (+ -1 q)          <--  r
                      (* q 1)           <--  f
                      (* q (+ -1 q) 1) <-- res
                      1                 <-- done
                      (* q (+ -1 q) 1) <-- res+
                      1)                <-- done+
                v))
```

**Si** $q > 3$ **:** Dans ce cas, le système fait 3 cycles de calcul. Le calcul de la factorielle pour $q > 3$ ne pouvant être complet (car après 3 cycles de calcul, on a $q >= 1$, si $q = 1$ le système a besoin d'un cycle supplémentaire pour terminer le calcul), le signal de sortie `res` n'est pas affecté. Cependant, les variables `r` et `f` nous renseignent sur le bon déroulement de l'étape de calcul, c'est-à-dire pour `r` : $q - 3$ et pour `f` : $q * (q - 1) * (q - 2)$.

```
(implies (and (integerp q)
              (<= 0 q)
              (not (equal q 1))
              (not (equal (+ -1 q) 1))
              (not (equal (+ -2 q) 1))
              (not (equal (+ -3 q) 1)))
         (equal (list q 1 (+ -3 q)
                      (* q (+ -1 q) (+ -2 q) 1)
                      (* q (+ -1 q) (+ -2 q) 1)
                      1 0 (+ -3 q)
                      (* q (+ -1 q) (+ -2 q) 1)
                      (* q (+ -1 q) (+ -2 q) (+ -3 q) 1)
```

```
                1 1 2
                (+ -3 q)                    <-- r
                (* q (+ -1 q) (+ -2 q) 1) <-- f
                0                           <-- res
                0                           <-- done
                0                           <-- res+
                0)                          <-- done+
        v))
```

### 4.2.2 Simsym

Nous avons repris et transformés les sources de la bibliothèque "expander.lisp" développés par Matt Kaufmann [58] : La commande `symsim` implémentée dans cette bibliothèque, opère une simplification de termes. Le principe est exactement le même que celui mentionné ci-dessus, à la différence que le terme n'est pas mis dans un théorème mais est envoyé directement au simplificateur d'Acl2 et est affiché sans appliquer d'autres heuristiques. Notre contribution est de générer une commande `Symsim` adaptée à notre modèle VHDL.

La syntaxe de la commande est la suivante :

```
(symsim appel_de_la_fonction_à_simuler
        (hypothèses
        ))
```

Par exemple, pour une simulation symbolique de la factorielle :

```
ACL2 !>(symsim (fact-simul 12 st)
               ((equal st (fact-make-state :arg q :start 1))
                (integerp q) (>= q 0)))
```

L'amélioration de cette bibliothèque a consisté à :
- Transformation du format de sortie,
- Ajout d'un canal de sortie : redirection des résultats vers un fichier texte,
- Commandes spécifiques `make-symsim` et `make-symsim-fn`, créant des appels à symsim spécifiquement pour les modèles VHDL.

### 4.2.3 Succession d'appels des accesseurs de l'état-mémoire

La pratique de la simulation symbolique expliquée dans les sections 4.2.1 et 4.2.2 est lente et devient vite difficile à lire lorsque l'on traite des exemples industriels où l'état-mémoire peut comporter plusieurs centaines de valeurs.
La simulation symbolique nous permet d'avoir une expression des sorties en fonction des entrées. Cependant, il peut être judicieux de posséder la succession d'appels `fact-putst` et `fact-getst` agissant sur l'état initial. En effet, cette succession d'appels indique seulement les modifications (et non pas l'état-mémoire total) apportés à un état initial, et l'expression obtenue peut être utilisée pour les preuves.

Pour cela, nous devons désactiver les définitions de `fact-putst` et `fact-getst`, de la façon suivante : `ACL2 !>(in-theory (disable fact-putst fact-getst))` Ajoutons les règles de simplifications suivantes :

Simplifications de la succession d'appels fact-getst et fact-putst:

P1: `(fact-getst 'vari (fact-putst 'vari a st))` = `a`
*Si la variable vari reçoit la valeur a et qu'ensuite on récupère sa valeur, on récupère a.*

P2: `vari` $\neq$ `varj` $\rightarrow$
`(fact-getst 'vari (fact-putst varj a st))` = `(fact-getst 'vari st)`
*La modification d'une variable varj n'a pas d'effets sur la valeur de la variable vari.*

Simplifications de la succession d'appels de fact-putst:

P3: `(fact-putst vari a (fact-putst vari b st))` = `(fact-putst vari a st))`

Arrangements des appels dans l'ordre des déclarations des signaux et variables:

P4: `(fact-get-nth varj)` < `(fact-get-nth vari)` $\rightarrow$
`(fact-putst vari a (fact-putst varj b st))` =
`(fact-putst varj b (fact-putst vari a st))`

Nous obtenons une *forme normale*, *unique et ordonnée* (sur les noms de variable dans l'ordre de leur déclaration dans `fact-get-nth`) , d'appels d'accesseurs à l'état mémoire. C'est-à-dire non pas l'état mémoire après 12 cycles de simulation mais les *modifications* qui sont faites sur ces 12 cycles. Les éléments mémoires non modifiés ne sont alors pas représentés.

Pour l'exemple de la factorielle, le résultat de simulation symbolique des accesseurs et présenté figure 4.3 avec 12 cycles de simulation et $q > 3$.

Dans la méthode précédente, nous avons vu que le but pouvait se diviser en sous-buts lorsque des "séparations de cas" pouvaient survenir, comme par exemple sur la valeur de $q$ si $q <= 3$. Or, l'avantage de la commande symsim est de ne pas utiliser les théorèmes, la séparation en "sous-buts" se faisant par l'ajout de la commande `cond` sur le résultat de simulation symbolique.

Par exemple :

```
(cond
  ((equal (nth 0 st) '1)       ; si q = 1
          (fact-putst
            'r 1
            (fact-putst
              'f 1
               (fact-putst
                 ...   ))))
  ((equal (nth 0 st) '2)       ; si q = 2
          (fact-putst
            'resmult (nth 0 st)
              (fact-putst
```

```
                                                  Commentaires :
                                        ; Element à l'instant t <- Element à l'instant t-1

(fact-putst                                                ; op1 <- op1+
   'op1 (+ -3 (fact-getst 'q st))
   (fact-putst                                             ; op2 <- op2+
    'op2 (* (fact-getst 'q st)
            (+ -1 (fact-getst 'q st))
            (+ -2 (fact-getst 'q st)) 1)
    (fact-putst                                            ; resmult <- resmult+
     'resmult (* (fact-getst 'q st)
                 (+ -1 (fact-getst 'q st))
                 (+ -2 (fact-getst 'q st)) 1)
     (fact-putst                                           ; startmult <- startmult+
      'startmult 1
      (fact-putst                                          ; endmult <- endmult+
       'endmult 0
       (fact-putst                                         ; op1+ <- q-3
        'op1+ (+ -3 (fact-getst 'q st))
        (fact-putst                                        ; op2+ <- q*(q-1)*(q-2)
         'op2+ (* (fact-getst 'q st)
                  (+ -1 (fact-getst 'q st))
                  (+ -2 (fact-getst 'q st)) 1)
         (fact-putst                                       ; resmult+ <- q*(q-1)*(q-2)*(q-3)
          'resmult+ (* (fact-getst 'q st)
                       (+ -1 (fact-getst 'q st))
                       (+ -2 (fact-getst 'q st))
                       (+ -3 (fact-getst 'q st)) 1)
          (fact-putst                                      ; startmult+ <- 1
           'startmult+ 1
           (fact-putst                                     ; endmult+ <- 1
            'endmult+ 1
            (fact-putst                                    ; mystate <- 2
             'mystate 2
             (fact-putst                                   ; r <- q-3
              'r (+ -3 (fact-getst 'q st))
              (fact-putst                                  ; f <- q*(q-1)*(q-3)
                 'f (* (fact-getst 'q st)
                       (+ -1 (fact-getst 'q st))
                       (+ -2 (fact-getst 'q st)) 1)
                 (fact-putst                               ; res <- res+
                      'res 0
                      (fact-putst                          ; done <- done+
                           'done 0
                           '((q 1 0 0 0 0 0 0 0 0 0
                                0 0 0 1 0 0 0 0)
                             ))))))))))))))))
```

FIG. 4.3 – *Simulation symbolique de la factorielle après 12 cycles de simulation avec* $q > 3$

```
              'startmult 1
                (fact-putst
                  'endmult (* (nth 0 st) 1)
            ...))))
  ((equal (nth 0 st) '3)       ; si q = 3
           ...)
  ((> (nth 0 st) '3)           ; si q > 3
          ...   ))
```

### 4.2.4 Performances de la simulation symbolique

La simulation symbolique est un "dépliage" de fonctions qui recherche, dans la totalité de la base de données de règles d'Acl2, celles applicables pour simplifier les expressions obtenues. En pratique, approximativement 900 000 règles doivent être triées par Acl2 lors de toutes tentatives de preuve en général. Lorsque nous effectuons une simulation symbolique sur plusieurs cycles de simulation, l'exécution devient lente car le nombre d'appels aux fonctions "à déplier et à simplifier" croît exponentiellement. Pour donner une idée du temps, une simulation symbolique pour 12 cycles de simulation de notre factorielle prend environ 20 secondes(Sun 450 Mhz).

Pour améliorer l'exécution, des règles de réécriture plus efficaces (notamment [40]) mais surtout l'algorithme $\nu$-rewrite de J Moore [62] offrent une complexité linéaire dans les simplifications des appels *fact-getst* et *fact-putst*. De plus, couplé avec l'utilisation d'une mémoire cache, ce nouveau moteur de réécriture améliore sensiblement la vitesse d'un rapport de 70 à 10000[62]. Ce nouveau moteur de réécriture sera disponible dans la future version d'Acl2 (2.6).

# Chapitre 5

# Preuves sur les résultats de simulations symboliques

Nous allons détailler dans ce chapitre quelques méthodes de preuve basées sur les résultats de la simulation symbolique. Rappelons que les résultats de simulation symbolique, couplés avec les règles de réécritures, permettent d'obtenir une *forme normale*, *unique et ordonnée*, d'appels d'accesseurs à l'état mémoire. Avec cette forme simplifiée de la description, nous illustrons quelques applications théoriques et pratiques pour effectuer des preuves d'équivalence et des preuves de propriétés, incluant des preuves par induction.

## 5.1 Preuves de propriétés sur les modèles Acl2

### 5.1.1 Type de propriétés (nombre de cycles d'horloge fixés)

La simulation symbolique d'un modèle VHDL sur plusieurs cycles de simulation ne possède pas toujours une forme pratique à lire et à comprendre, de plus elle n'exprime pas toujours, pour un nombre de cycles fixé, une propriété lisible par l'utilisateur. Celui-ci désirant valider son implémentation peut exprimer des propriétés sur le modèle. Ces propriétés sont établies par rapport à un état donné de référence sur lequel est appliqué un nombre fixé de cycles d'horloge. Les schémas usuels de ces types de propriété sont les suivants :

**Type de propriété 1 :** À partir de l'état $st$ créé par (*arch-entity*-make-state ...), la valeur de l'élément $var$ de l'état-mémoire est égale à $resultat$ après $x$ cycles de simulation.

```
(defthm Propriete_sur_le_modele
        (implies (and (arch-entity-statep st)
                      (equal st (arch-entity-make-state ..< valeurs >))
                      < ... Hypothèses supplémentaires ... >)
        (equal (arch-entity-getst 'var (arch-entity-simul x st))
               (<résultat >))))

où <résultat> est une expression.
```

**Type de propriété 2 :** À partir de l'état $st$ créé par (*arch-entity*-make-state ...), la valeur de la fonction $resultatp$ sur l'élément $var$ de l'état-mémoire est égale à t (vrai) après $x$ cycles de simulation.

(**defthm** *Propriete_sur_le_modele*
(implies (and (*arch-entity*-statep st)
(equal st (*arch-entity*-make-state ..< *valeurs* >))
< ... *Hypothèses supplémentaires* ... >)
(*résultatp* (*arch-entity*-getst 'var (*arch-entity*-simul $x$ st)))))

où *résultatp* est une fonction booléenne.

Il est également possible de mélanger les deux schémas pour prouver que l'application d'une fonction $f$ à un résultat donné est égale à un autre résultat.

Exemple de propriété:

**Théorème** : À partir de l'état $st$ où la valeur de $arg$ est fixée à un entier $q$ tel que $q > 4$, la valeur du signal $resmult$ après 12 cycles de simulation est égale à (* q (+ -1 q) (+ -2 q) (+ -3 q)).

```
(defthm Propriete_sur_factorielle
        (implies (and (fact-statep st)
                      (equal st (fact-make-state :arg q :start 1))
                      (integerp q) (> q 4))
        (equal (fact-getst 'resmult (fact-simul 12 st))
               (* q (+ -1 q) (+ -2 q) (+ -3 q)))))
```

### 5.1.2 Applications : preuves d'équivalence de bloc d'instructions

Considérons les ensembles d'instructions suivants :

```
Main1:Process
 variable mem : bit_vector;
          adr1: natural;
         x,a,b: bit;
begin
wait until CLK ='1';
mem(adr1) := a;
x := mem(adr2);
res := x + b;
end process
```

```
Main2:Process
variable mem : bit_vector;
          adr1: natural;
         x,a,b: bit;
begin
 wait until CLK ='1';
 x := mem(adr2);
 mem(adr1) := a;
 IF adr1 = adr2
   THEN res := a + b;
   ELSE res := x + b;
 END IF
end process
```

Cet exemple a été proposé par Gerd ritter, qui n'a pas pu montrer cette équivalence dans son simulateur symbolique [71, 72]. L'instruction IF dans le process Main2 divise l'ensemble d'instructions en deux sous-ensembles : l'un avec la condition $adr1 = adr2$ vraie, l'autre faux. Pour prouver l'équivalence, Acl2 compare les deux sous-ensembles obtenus avec les instructions du process Main1. Les deux process sont inclus dans une architecture "arch" qui dispose d'une entité "ent" vide. La description est modélisée en Acl2.

Le théorème à montrer est le suivant:

**Théorème** : Le bloc d'instructions séquentielles représenté dans le process Main1 est équivalent au bloc d'instructions séquentielles représenté dans le process Main2.

```
(defthm Equivalence_instructions
        (implies (and (equal st (arch-ent-make-state))
                      (arch-ent-statep st))
                 (equal (main1 st)
                        (main2 st))))
```

Considérons maintenant les 2 blocs d'instructions ci-dessous :

```
main1:process
 variable var1,var2:natural;
 variable in2,in3:natural;
 variable out1:natural;
begin
 wait until clk ='1';
 var1 := var2 - in2;
 var2 := var1 * in3;
 out1 := in3 + in2;
```

```
main2:process
 variable var1,var2:natural;
 variable in2,in3:natural;
 variable out1:natural;
begin
 wait until clk ='1';
 var1 := in1 * var2;
 var2 := in1 + in3;
 out1 := in2;
```

Utilisant une méthode similaire, nous montrons que la composition séquentielle de ces séquences d'instructions est équivalente à la séquence d'instructions suivante:

```
main3:process
 variable var1,var2:natural;
 variable in2,in3:natural;
 variable out1:natural;
begin
 wait until clk ='1';
 var1 := (in1 + in3) - in2;
 var2 := in3 * (in1 * var2);
 out1 := in3 + in2;
```

L'exemple était proposé par Julia Dushina [36] pour vérifier l'équivalence entre deux machines d'états abstraites, lors de la vérification de l'étape d'ordonnancement d'un outil effectuant de la synthèse de haut niveau.

Cette méthode peut être étendue afin de prouver l'équivalence de deux sorties entre deux modèles VHDL (utilisant des états-mémoires différents) après un nombre de cycles de simulation (qui peut être différent pour chaque modèle). Voici le schéma global:

**Théorème** :Les modèles *arch-entity1* et *arch-entity2* calculent la même expression sur les sorties respectives $S_1$ et $S_1'$ après respectivement $n_1$ et $n_2$ cycles de simulation du modèle.

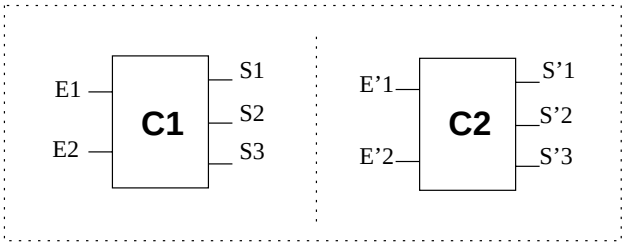


FIG. 5.1 – *Représentation de C1 et C2*

```
(defthm Equivalence_d'architectures
        (implies (and (equal st1 (arch-entity1-make-state < valeurs >))
                      (equal st2 (arch-entity2-make-state < valeurs >))
                      < ... Correspondances ...>)
        (equal (arch-entity1-getst 'S1 (arch-entity1-simul n1 st1))
               (arch-entity2-getst 'S'1 (arch-entity2-simul n2 st2)))))
```

Pour illustrer le principe de la vérification d'équivalence, nous considérons deux descriptions de circuits, appelons-les $C_1$ et $C_2$, ayant comme état mémoire respectivement $st_1$ et $st_2$. Les descriptions de ces circuits comme "boîtes noires" sont représentées sur la figure 5.1. Prouver que $C_1$ et $C_2$ sont *fonctionnellement* équivalents se résoud, dans notre système, à comparer les expressions symboliques de sorties respectivement entre elles. *Correspondances* relient les positions des entrées de l'état mémoire $st_1$ avec les positions des entrées de l'état mémoire $st_2$.

Par exemple, supposons que la sortie $S_1$ du composant $C_1$ (couple arch/entity1) donne comme expression algébrique:
(* (arch-entity1 'E1 $st_1$) (- (arch-entity1 'E2 $st_1$) (arch-entity1 'L $st_1$))) (L étant un signal local)
et que la sortie $S_2$ du composant $C_2$ (couple arch/entity2) donne en sortie
(- (* (arch-entity2 'L' $st_2$) (arch-entity2 'E'1 $st_2$)) (* (arch-entity2 'L' $st_2$) (arch-entity 'E'2 $st_2$)))
(L' étant un signal local)
.

Acl2 a besoin, pour établir l'équivalence, que les éléments
E1, E2, L
correspondent respectivement aux éléments
L', E'1 et E'2.
Ce qui s'écrit dans le théorème précédent :

```
(equal (arch-entity1 'E1 st1)
       (arch-entity2 'L' st2))
(equal (arch-entity1 'E2 st1)
       (arch-entity2 'E'1 st2))
(equal (arch-entity1 'L st1)
       (arch-entity2 'E'2 st2))
```

Pour démontrer cette propriété d'équivalence, Acl2 a seulement besoin de la propriété de distributivité de la multiplication.

## 5.2 Preuves de propriétés inductives

### 5.2.1 La factorielle de n

Reprenons l'exemple de notre factorielle (figure 3.2). Le but étant de prouver que la description réalise bien une factorielle, après un nombre de cycles de simulation qui dépend linéairement de l'entrée `arg`, c'est-à-dire que sa spécification ci-dessous donnée en Lisp :

```
(defun factorial (n)
   (if (zp n)
          1
          (* n (factorial (1- n)))))
```

correspond au comportement attendu de notre modèle.

Pour valider notre implémentation, nous avons eu recours à deux approches. La première consiste à donner des théorèmes sur les fonctions lisp de la description. La seconde, basée sur les résultats de simulation symbolique, est plus courte et surtout automatisable.

**Théorèmes sur les fonctions**

L'approche, illustrée dans [11], consiste en une série de théorèmes qui décrivent les moindres modifications de chaque fonction sur l'état mémoire, à partir de chaque affectation jusqu'aux process et au cycle de simulation.

Ces théorèmes peuvent être divisés en deux classes :

**Conservation des propriétés sur l'état mémoire :** Chaque fonction qui prend en argument un état mémoire `st` et retourne un nouvel état mémoire `st'` est telle que, si `st` est une liste, c'est-à-dire que l'appel `(consp st)` est évalué à `T`, le résultat retourné `st'` est aussi une liste, et sa longueur est égale à celle de `st`. Ces propriétés garantissent le typage des fonctions manipulant les états-mémoires.

Par exemple, pour la fonction `fact-putst` :

```
(defthm fact-putst_consp
        (implies (consp st)
                 (consp (fact-putst var new_value st))))

(defthm length_fact-putst
        (equal (len (fact-putst var new_value st))
               (len st)))
```

Ces propriétés sont prouvées pour les fonctions suivantes:
`fact.doit-cycle`, `fact.multiplier-cycle`, `fact-cycle`, dans cette ordre.

**Variables de l'état mémoire modifiées ou inchangées :** Pour certaines fonctions, un certain nombre d'éléments de l'état mémoire ne change pas pendant leur exécution. Ces propriétés d'invariance, utilisées comme règles de réécritures, permettent des simplifications supplémentaires pour les preuves ultérieures. Il est donc nécessaire pour l'utilisateur d'identifier lui-même le comportement de chaque fonction et de lui associer ses propriétés.

Voici un exemple :

**Théorème** : Les éléments suivants de l'état mémoire ne sont pas affectés par le déroulement du process Doit : arg start op1 op2 resmult startmult endmult resmult+ endmult+ res done. Si *var* est un élément de cette liste, (fact-getst 'var (fact.doit-cycle st)) se réécrit en (fact-getst 'var st).

```
(defthm fact.doit-cycle_not_modified
        (implies (and (equal (len st) 19)
                      (member var '('arg 'start 'op1 'op2 'resmult 'startmult
                                    'endmult 'resmult+ 'endmult+ 'res 'done)
                 (equal (fact-getst 'var (fact.doit-cycle st))
                        (fact-getst 'var st))))))
```

Des propriétés sur le cycle de calcul sont établies sur certains éléments de l'état mémoire, tel que sur la variable `r`.

**Théorème** : La valeur de la variable $r$ après un cycle de simulation est égale à:
- resmult si la valeur de endmult était égale à 1
et mystate était égale à 2 au cycle précédent,
- 1 si la valeur de mystate était égale à 0
- la valeur de r du cycle précédent sinon.

```
(defthm lemma_fact-cycle7
        (implies (equal (len st) 19)
                 (equal (fact-getst 'r (fact-cycle st))
                        (if (equal (fact-getst 'mystate st) 2)
                            (if (equal (fact-getst 'endmult st) 1)
                                (fact-getst 'resmult st)
                                (fact-getst 'r st))
                            (if (equal (fact-getst 'mystate st) 0)
                                1
                                (fact-getst 'r st))))))
```

Nous définissons ensuite une fonction nommée `execute` qui exécute la totalité des cycles de calcul (nous rappelons qu'un cycle de calcul représente 3 cycles de simulation, figure 5.2) requis pour avoir la factorielle de l'entrée. `execute` retourne l'état-mémoire final. Cette fonction récursive nécessite de connaître les informations suivantes :
- Quel est l'élément de l'état-mémoire qui possède une mesure qui décroit?
- Quel est le critère d'arrêt par rapport à cet élément?

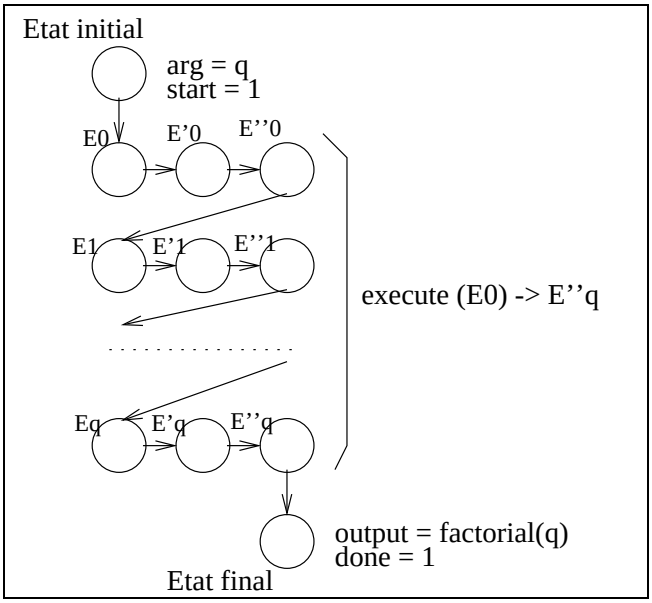


FIG. 5.2 – *Fonction execute*

Ces informations constituent le schéma d'induction utilisable pour la preuve ultérieure de la factorielle.

- Quels sont les invariants entre deux pas de calcul?

Il est important ici de noter que nous ne nous situons pas à l'état initial (l'état d'initialisation) mais à un cycle plus tard. En effet, comme l'illustre la figure 4.1 (chaque transition est un cycle de simulation), nous utilisons une symétrie des états de contrôle. Nous verrons plus loin comment ramener les preuves à l'état initial et à l'état final. Voici la définition récursive de la fonction `execute` :

```
(defun execute (st)
(declare (xargs :measure (nfix (fact-getst 'r st))
                :hints (("Goal" :in-theory (disable fact-simul))))
(if (and (integerp (fact-getst 'r st))
         (equal (len st) 19)
         (equal (fact-getst 'mystate st) 1)
         (equal (fact-getst 'startmult st) 0))
   (if (< (fact-getst 'r st) 2)
      st
   (execute (fact-simul 3 st)))
   st))
```

Les indications fournies dans la partie `declare` de la définition de la fonction servent pour l'admission de `execute`: l'indication de la "mesure" et le "masquage" de la définition de fact-simul.

Le fichier `old.fact-proof.lisp` se trouve en annexe, il contient le théorème final, obtenu

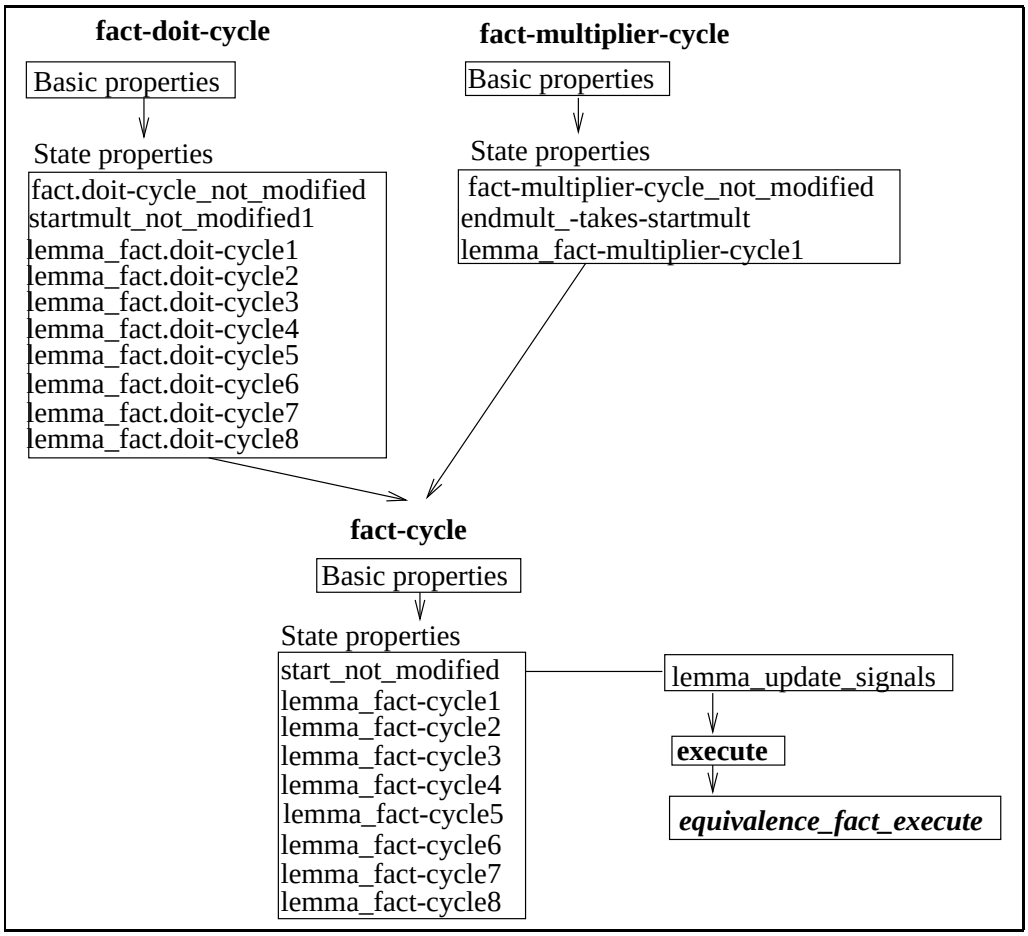


FIG. 5.3 – *Structure des théorèmes*

avec 39 lemmes (la structure des théorèmes importants est représentée sur la figure 5.3). La certification du fichier, c'est-à-dire la preuve de tout les théorèmes, prend environ 22 secondes sur un Sun 450 Mhz.

**Théorème** : La valeur de retour de la fonction execute est égale à la factorielle de la valeur de $r$, multipliée par la valeur initiale de $f$ (si celle-ci n'est pas fixée à 1).

```
(defthm equivalence_fact_execute
        (implies (and (true-listp st)
                      (integerp (fact-getst 'r st))
                      (equal (len st) 19)
                      (equal (fact-getst 'mystate st) 1)
                      (equal (fact-getst 'startmult st) 0)
                      (<= 2 (fact-getst 'r st)))
                 (equal (fact-getst 'f (execute st))
                        (* (fact-getst 'f st)
                           (factorial (fact-getst 'r st)))))
        :hints ...)
```

### Une méthode basée sur la simulation symbolique

La méthode présentée ici utilise les résultats de simulation symbolique. Pour réutiliser ces résultats, ils sont exportés dans un fichier puis retraités pour former une nouvelle règle de réécriture. Voici l'exemple de réutilisation d'une simulation symbolique d'un cycle de calcul (soit 3 cycles de simulation) pour l'architecture `Fact` :

```
(defthm rewrite_one_computation_step
        (implies (and (true-listp st)
                      (integerp (fact-getst 'r st))
                      (equal (len st) 19)
                      (equal (fact-getst 'mystate st) 1)
                      (equal (fact-getst 'startmult st) 0)
                      (<= 2 (fact-getst 'r st)))
                 (equal (fact-simul 3 st)
                        < ... Résultats de simulation symbolique ... )))>
```

Ainsi, cette règle remplace toute expression `(fact-simul 3 st)` par l'expression symbolique si `st` satisfait aux hypothèses.

La définition de la fonction `execute` :

```
(defun execute (st)
       (declare (xargs :measure (nfix (fact-getst 'r st))
                       :hints (("Goal"
                                    :in-theory (disable fact-simul)))))
       (if (and (true-listp st)
                (equal (len st) 19)
                (integerp (fact-getst 'f st))
                (integerp (fact-getst 'r st)) (< 0 (fact-getst 'r st))
                (equal (fact-getst 'startmult st) 0)
                (equal (fact-getst 'start st) 1)
                (equal (fact-getst 'mystate st) 1))
           (if (< (fact-getst 'r st) 2)
               st
               (execute (fact-simul 3 st)))
           st))
```

Et ainsi, directement le théorème final :

**Théorème** : La valeur de retour de la fonction execute est égale à la factorielle de la valeur de $r$, multipliée par la valeur initiale de $f$ (si celle-ci n'est pas fixée à 1).

```
(defthm Equivalence_fact
        (implies (and (true-listp st)
                      (integerp (fact-getst 'f st))
                      (integerp (fact-getst 'r st))
                      (<= 1 (fact-getst 'r st))
                      (equal (len st) 19)
                      (equal (fact-getst 'startmult st) 0)
                      (equal (fact-getst 'start st) 1)
                      (equal (fact-getst 'mystate st) 1)
                 )
                 (equal (fact-getst 'f (execute st))
                        (* (fact-getst 'f st)
                           (factorial (fact-getst 'r st)))))
        :hints (("Goal" :in-theory (disable fact-simul fact-cycle fact-getst))))
```

Il est important de désactiver les définitions des fonctions fact-simul, fact-cycle et fact-getst afin que seules les règles de réécritures soient employées. La certification du fichier prend 1.65 secondes et ne contient pas de lemmes intermédiaires.

### 5.2.2 La multiplication

Afin de conforter la méthodologie, nous allons utiliser un autre exemple. Il s'agit d'une description VHDL réalisant une multiplication par addition itérée (Figure 5.4). La formalisation de cette description est présentée en annexe.

Le comportement est modélisé sur la figure 5.5, chaque fonction de transition modélisant un cycle de simulation. La méthode de preuve utilise comme précédemment les résultats de simulation symbolique obtenue sur l'architecture `Mult` (cf annexe).

```
entity MULT is
  port (A,B: in NATURAL; REQ,CLK: in BIT; C: out NATURAL; DONE: out BIT);
end MULT;
architecture BEHAV of MULT is
begin
  MULTIPLIER: process
   VARIABLE mult-state,prod,count: NATURAL;
  begin
   wait until CLK = '1';
   case mult-state is
    when 0 =>
      if REQ = '1' then
        prod := 0;
        count := A;
        mult-state := 1;
      end if;
    when 1 =>
      if REQ = '0' then
        mult-state := 0;
      else
         if (count > 0) then
           prod := prod+B;
           count := count-1;
         else
           C <= prod;
           DONE <= '1';
           mult-state := 2;
         end if;
      end if;
    when 2 =>
      if REQ = '0' then
         C <= 0;
         done <= '0';
         mult-state := 0;
      end if;
  end case;
  end process;
end BEHAV;
```

FIG. 5.4 – *Multiplication par addition itérée*

La fonction `execute` possède la définition suivante :

```
(defun execute (st)
       (declare (xargs :measure (acl2-count (mult-getst 'count st))
                       :hints (("Goal" :in-theory (disable mult-cycle)))))
       (if (and (true-listp st)
                (equal (mult-getst 'req st) 1)
                (mult-statep st)
                (equal (mult-getst 'mult-state st) 1))
           (if (zp (mult-getst 'count st))
               st
               (execute (mult-cycle st)))
           st))
```

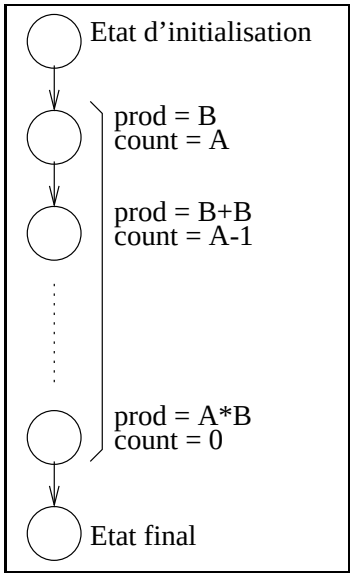


FIG. 5.5 – *Modélisation des cycles de calcul de Mult*

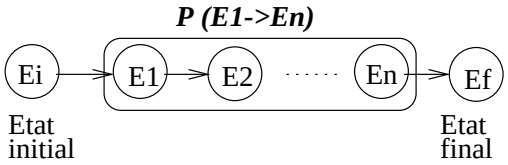


FIG. 5.6 – *Décalage de propriétés*

Le théorème établissant la preuve de la multiplication :

**Théorème :** La variable prod contient $(count * b) + prod$

```
(defthm correctness_of_execute
        (implies (and (true-listp st)
                      (equal (mult-getst 'req st) 1)
                      (mult-statep st)
                      (equal (mult-getst mult-state st) 1)
                      (>= (mult-getst 'count st) 0)
                      (>= (mult-getst 'b st) 1))
                 (equal (mult-getst 'prod (execute st))
                        (+ (* (mult-getst 'count st)
                              (mult-getst 'b st))
                           (mult-getst 'prod st))))
:hints (("Goal" :in-theory (disable mult-cycle))))
```

; Q.E.D 1.27 sec

### 5.2.3 Etat d'initialisation et état final

Les preuves effectuées sur la description factorielle et multiplication se basent sur les cycles de calcul et omettent le premier et dernier cycle de simulation. Pour ramener la preuve effectuée sur les cycles de calcul de la factorielle à l'état d'initialisation et à l'état final, il faut effectuer

des "décalages" de propriétés. Appelons **P(E1 -> En)** une propriété de la description à partir d'un état E1 jusqu'à un état En (figure 5.6). Le but étant de "décaler" la propriété pour qu'elle puisse s'exprimer à partir de l'état Ei jusqu'à l'état Ef.
L'opération se déroule en deux étapes :
- Transformer d'abord P(E1->En) en P'(Ei->En),
- puis en dériver P"(Ei->Ef).

Pour ramener la propriété de la multiplication à l'état initial et ensuite à l'état final, nous prouvons les théorèmes suivants :

**Théorème point de départ.** : Validation de la multiplication avec remplacement de la fonction *execute* par la fonction $mult - simul$ : la valeur de *prod* après *count* cycles de simulation est égale à la multiplication de *count* par *b*, additionnée par le contenu initial de *prod*.

```
(defthm theorem_mult-simul_is_multiplication
        (implies (and (true-listp st)
                      (mult-getst 'req st)
                      (mult-statep st)
                      (equal (mult-getst 'mult-state st) 1)
                      (>= (mult-getst 'count st) 1)
                      (>= (mult-getst 'b st) 1))
                 (equal (mult-getst 'prod
                                    (mult-simul (mult-getst 'count st) st))
                        (+ (* (mult-getst 'count st) (mult-getst 'b st))
                           (mult-getst 'prod st))))

:hints (("Goal" :in-theory (disable mult-simul mult-cycle)
                :use correctness_of_execute)))
```

Ce théorème est la propriété P(E1->En), il s'agit d'abord de translater en P'(Ei->En), grâce au théorème 1 ci-dessous. Auparavant un théorème intermédiaire doit formaliser le décalage d'état :

**Théorème intermédiaire** : À partir de l'état initial (où la variable *mystate* est à 0), la valeur de la variable *prod* après un cycle de simulation, puis *count* cycle de simulation, est égale à la multiplication de *count* par *b*.

```
(defthm total_result_prod
        (implies (and (true-listp st0)
                      (mult-statep st0)
                      (mult-getst 'req st0)
                      (equal (mult-getst 'mult-state st0) 0)
                      (>= (mult-getst 'count st0) 1)
                      (>= (mult-getst 'b st0) 1))
                 (equal (mult-getst 'prod
                                    (mult-simul
                                                (mult-getst 'count (mult-cycle st0))
                                                (mult-cycle st0)))
                        (* (mult-getst 'count (mult-cycle st0))
                           (mult-getst 'b st0))))
:hints (("Goal" :do-not-induct t
                :in-theory (disable mult-cycle mult-simul
                                    mult-unfold rewrite_computation_step)
                :use (:instance
                                theorem_mult-simul_is_multiplication (st (mult-cycle st0))))))
```

Le *hint* `:use ...` a pour but de réutiliser le théorème mis en argument dans la partie hypothèse du nouveau théorème. Ici nous demandons à Acl2 d'utiliser le théorème `theorem_mult-simul_is_multiplication` en remplaçant l'état mémoire `st` dans ce théorème par l'état mémoire de l'état initial `st0` dont on applique un cycle de simulation, soit `(mult-cycle st0)`.

**Théorème 1** : Validation de la multiplication par utilisation des ports d'entrée, $a$ et $b$, soit $prod = a * b$.

```
(defthm total_result_prod2
        (implies (and (true-listp st0)
                      (mult-statep st0)
                      (mult-getst 'req st0)
                      (equal (mult-getst 'mult-state st0) 0)
                      (>= (mult-getst 'a st0) 1)
                      (>= (mult-getst 'b st0) 1))
                 (equal (mult-getst 'prod (mult-simul
                                                (mult-getst 'a st0)
                                                (mult-cycle st0)))
                        (* (mult-getst 'a st0) (mult-getst 'b st0))))
:hints (("Goal" :do-not-induct t
                :in-theory (disable mult-cycle mult-simul
                                    mult-unfold rewrite_computation_step)
                :use total_result_prod)))
```

Le théorème final, P"(Ei->Ef) est donné ainsi:

**Théorème** : Validation complète de la multiplication

: $c = a * b$ après un cycle de simulation, puis $a$ cycles, puis un dernier cycle de simulation.

```
(defthm total_result_c
        (implies (and (true-listp st1)
                      (mult-statep st1)
                      (mult-getst 'req st1)
                      (equal (mult-getst 'mult-state st1) 0)
                      (equal (mult-getst 'done st1) nil)
                      (>= (mult-getst 'a st1) 1)
                      (>= (mult-getst 'b st1) 1))
                 (equal (mult-getst 'c+ (mult-cycle (mult-simul (mult-getst 'a st1)
                                                                (mult-cycle st1))))
                        (* (mult-getst 'a st1)
                           (mult-getst 'b st1))))
:hints (("Goal" :do-not-induct t
                :in-theory (disable mult-cycle mult-simul
                                    mult-unfold
                                    rewrite_computation_step)
                :use ((:instance total_result_prod2 (st (mult-simul (mult-getst 'a st1)
                                                                    (mult-cycle st1))))))))
```

### 5.2.4 Methodologies associées

L'expression obtenue par simulation symbolique montre les différents appels aux accesseurs de l'état-mémoire. La nouvelle règle de réécriture remplaçant la fonction *arch-entity*-cycle possède le schéma suivant (*step* étant le nombre de cycles de simulation pour effectuer un cycle de calcul.

```
(defthm rewrite_one_computation_step
        (implies (and (arch-entity-statep st)
                      < Invariants du cycle de calcul >)
        (equal (arch-entity-simul step st)
               < Résultat de simulation symbolique >)))
```

La méthodologie associée aux exemples précédents consiste à modéliser le cycle de calcul par une fonction récursive (ici, `execute`). L'acceptation de cette fonction par Acl2 entraîne la désignation d'une "mesure" qui décroît dans la récursion et une condition d'arrêt. Dans l'exemple de la factorielle, la variable `r` représente cette mesure et dans l'exemple de la multiplication, il s'agissait de `count`. Ces variables servent également de condition d'arrêt lorsqu'elles sont égales à 0. Cette information est primordiale pour le démonstrateur pour identifier le schéma d'induction du théorème final.

Voici le schéma de la fonction `execute` :

```
(defun execute
       (declare (xargs :measure (acl2-count (arch-entity-getst 'var st))
                       :hints (("Goal" :in-theory (disable arch-entity-simul)))))
       (if (and (true-listp st)
                (arch-entity-statep st)
                < ... Invariants des cycles de calcul ... >
           (if (zp (arch-entity-getst 'var st))
               st
               (execute (arch-entity-simul step st)))
       st)))
```

Bien que dans certains cas simples, Acl2 puisse trouver par lui-même l'argument qui décroit, il est néanmoins souhaitable que l'utilisateur fournisse cette information explicitement.

Le théorème final va utiliser comme variable d'induction l'argument utilisé pour l'acceptation de `execute`.

Voici le schéma adopté pour le théorème final (output représente l'élément de l'état-mémoire accumulant le résultat):

```
(defthm correctness_of_execute
          (implies (and (true-listp st)
                        (arch-entity-statep st)
                        < ... Invariants des cycles de calcul ... >)
                   (equal (arch-entity-getst 'output+ (execute st))
                          < ... Résultat attendu ... >

:hints (("Goal" :in-theory (disable arch-entity-cycle))))))
```

Ce dernier théorème nécessite d'avoir, en général, un résultat attendu qui est sous la forme d'une expression algébrique, ou alors d'une fonction récursive, celle-ci devant avoir un schéma d'induction identique au cycle de calcul.

# Chapitre 6

# Réutilisabilité des composants et des propriétés

La réutilisabilité des preuves est un point crucial lors de l'utilisation d'un modèle VHDL compositionnel. Un composant réutilisable apporte un gain de temps lors de la conception. De plus, pouvoir exporter ses propriétés nous permet de gagner du temps sur les preuves de l'architecture englobante. Nous exposons ici une méthode permettant l'abstraction d'un composant de plusieurs manières. La première méthode consiste à exporter les propriétés prouvés d'un composant sur une architecture englobante en adoptant un appel procédural. La seconde consiste à caractériser une famille de composant par des propriétés communes et à prouver des propriétés réutilisables sur toutes les instances de cette famille.

## 6.1 Exportation de propriétés

Considérons la description de l'architecture `BEHAV` de `MULT` comme un composant. Un système utilisant un tel composant doit obéir à deux caractéristiques de modélisation. Tout d'abord, l'état mémoire doit inclure un élément qui contient l'état mémoire local au composant. Ensuite, chaque cycle de simulation du système doit activer le composant au travers de la fonction `mult.behav-cycle` sur son état mémoire.

Soit $st_m$ l'état mémoire du multiplieur. Soit $st_s$ l'état mémoire d'un système utilisant le multiplieur comme composant. Pour le décrire, nous utilisons les fonctions ci-dessous: - Les fonctions accesseurs du système: `system-getst` et `system-putst`,
- Les fonctions accesseurs du composant: `comp-getst` et `comp-putst`,
- L'appel *(system-getst 'mult.behav* $st_s$*)* extrait l'état mémoire local au composant multiplieur à partir de celui du système englobant.
- La fonction *(system-simul* $n$ $st_s$*)* représente la fonction de simulation du système. Cette fonction effectue $n$ cycles de simulation du système qui invoque également $n$ cycles de simulation de l'état mémoire du multiplieur.

La définition de la fonction system-simul est la suivante:

(**defun** system-simul (n $st_s$)

```
(if (zp n)
    st_s
    (system-simul (1- n)
                  (system-cycle
                        (system-update-signals st_s)))))
```

Nous voulons prouver que la fonction `system-simul` peut s'exprimer en fonction de la fonction `mult.behav-simul`. En effet, considérons une propriété de la forme suivante (avec comp le nom du composant, k un nombre fixé de cycles de simulation, `comp-putst` l'accesseur de l'état-mémoire, P une expression donnée par l'utilisateur s'exprimant sur un signal S) :

```
(equal (comp-putst 'S (comp-simul k st_m))
       P)
```

La propriété pouvant alors s'exporter dans l'architecture englobante de la façon suivante (nous considérons le cas où le signal de sortie S du composant est relié à un signal local ou de sortie A de l'architecture `system`):

Propriété: La valeur de A provenant de k cycles de simulation du système correspond à la valeur de A provanant de k cycles de simulation du composant comp.

```
(equal (system-getst 'A (system-simul k st_s))
       (system-getst 'A
                     (system-putst 'comp
                                   (comp-simul k (system-getst 'comp st)))))
```

D'où, après changement du signal A par la sortie S du composant dans le second terme:

```
(equal (system-getst 'A (system-simul k st_s))
       (comp-getst 'S
                   (comp-simul k st_m)))
```

qui devient: La valeur du signal A après k cycles de simulation du système correspond à P.

```
(equal (system-getst 'A (system-simul k st_s))
       P)
```

Hypothèses sur la description Pour effectuer l'exportation de propriétés, nous devons envisager l'appel au composant comme un *appel procédural*, c'est-à-dire que l'architecture envoie une requête "`start`=1" pour faire débuter le calcul au composant et attend l'accusé de réception "`done`=1" donné par le composant à la fin de son calcul. Ainsi, nous pouvons supposer que pendant le cycle de calcul, le composant A n'est pas affecté par l'architecture, les entrées ne changent pas. Dans toute la suite, nous nous situons dans la cas où l'architecture à envoyé la valeur 1 à start et attend le signal `done`.

Preuve :

Pour faciliter les notations, nous introduisons le raccourci suivant :

```
(defun system-mult.behav-cycle (st)
        (system-putst 'mult.behav
                      (mult.behav-cycle
                                  (system-getst 'mult-behav
                                                st))))

(defun system-mult.behav-simul (n st)
        (system-putst 'mult.behav
                      (mult.behav-simul n
                                  (system-getst 'mult-behav
                                                st))))
```

La fonction `system-simul` possède la définition suivante :

```
(system-simul n st)
=
(system-simul (1- n)
              (system-cycle
                      system-update-signals st))
```

Supposons que l'architecture englobante `system` contient un process `process1`. En développant la fonction soulignée `system-cycle`, nous obtenons la forme suivante :

```
(system-simul n st)
=
(system-simul (1- n)
              (seq st
                   (system-process1-cycle (system-update-signals st))
                   (system-mult.behav-cycle st)))
```

Réécrivons cette forme en changeant de notation et en montrant les successions d'appels:

```
(system-simul n st)
=
(system-simul (1- n)
              (system-mult.behav-cycle (system-process1-cycle (system-update-signals st))))
```

Ces appels peuvent se diviser en deux appels concurrents (car les entrées du composants ne sont pas modifiées par l'architecture `system` et les sorties n'affectent pas l'architecture) :

```
(system-simul n st)
=
```

```
(system-simul (1- n)
        (let ((st_1 (system-process1-cycle (system-update-signals st)))
            (st_2 (mult.behav-cycle
                        (system-getst 'mult.behav
                            (system-update-signals st))))
            (system-putst 'mult.behav st_2 st_1))))
```

Nous allons effectuer un raisonnement par induction pour montrer que system-simul peut s'écrire en fonction de mult.behav-simul.

**Raisonnement par induction**

**cas de base : n=1 :**

```
(system-simul 1 st)
=
(system-simul 0
        (let ((st_1 (system-process1-cycle (system-update-signals st)))
            (st_2 (mult.behav-cycle
                        (system-getst 'mult.behav
                            (system-update-signals st))))
            (system-putst 'mult.behav st_2 st_1))))
```

*Par expansion de la fonction soulignée :*

```
=
(system-simul 0
        (let ((st_1 (system-process1-cycle (system-update-signals st)))
            (st_2 (mult.behav-cycle
                        (system-getst 'mult.behav
                            (seq st
                                Mise à jour des signaux de system
                                (system-update-of-mult.behav st)
                                < Affectation des ports d'entrée de mult.behav >
                                < Transmission des valeurs des ports de sortie))))
            (system-putst 'mult.behav st_2 st_1))))
```

*L'élément souligné peut être supprimé car ne modifiant pas les éléments du composant.*

```
=
(system-simul 0
        (let ((st_1 (system-process1-cycle (system-update-signals st)))
            (st_2 (mult.behav-cycle
                        (system-getst 'mult.behav
                            (seq st
                            (system-update-of-mult.behav st)
                            < Affectation des ports d'entrée de mult.behav >
                            < Transmission des valeurs des ports de sortie >))))
            (system-putst 'mult.behav st_2 st_1))))
```

*d'où, en réunissant les éléments soulignés :*

```
=
(system-simul 0
              (let ((st_1 (system-process1-cycle (system-update-signals st)))
                    (st_2 (mult.behav-simul
                                        1
                                        (seq st
                                             < Affectation des ports d'entrée de mult.behav >
                                             < Transmission des valeurs des ports de sortie >))))
                    (system-putst 'mult.behav st_2 st_1)))
```

**Hypothèse pour n:**

```
(system-simul n st)
=
(system-simul 0
              (let ((st_1 (system-process1-cycle
                           (system-update-signals
                              (system-process1-cycle
                                 (system-update-signals
                              ... n appels))))
                    (st_2 (mult.behav-simul
                                        n
                                        (seq st
                                        < Affectation des ports d'entrée de mult.behav >
                                        < Transmission des valeurs des ports de sortie >))))
                    (system-putst 'mult.behav st_2 st_1))))
```

**Démonstration pour n+1:**

```
(system-simul (1+ n) st)
=
system-simul n
             (system-cycle
                  (system-update-signals st)))
devient par application de l'hypothèse d'induction :
(system-simul 0
              (let ((st_1 (system-process1-cycle
                           (system-update-signals
                                (system-process1-cycle
                                     (system-update-signals
                           ... n appels
                           (system-cycle
                            (system-update-signals st))))
                    (st_2 (mult.behav-simul
                              n
                              (seq st
                              (system-cycle (system-update-signals st))
                              < Affectation des ports d'entrée de mult.behav >
```

```
                                < Transmission des valeurs des ports de sortie >))))
                        (system-putst 'mult.behav st_2 st_1)))))
Par expansion des termes soilignés :
=
(system-simul 0
              (let ((st_1 (system-process1-cycle
                            (system-update-signals
                                 (system-process1-cycle
                                      (system-update-signals
                            . . . n appels
                                      (let ((st_1+ (system-process1-cycle
                                                        (system-update-signals st)))
                                           (st_2+ (mult.behav-simul
                                                  1
                                                  (seq st
                                                  < Affectation des ports d'entrée de mult.behav >
                                                  < Transmission des valeurs des ports de sortie > ))))
                                           (system-putst 'mult.behav st_2+ st_1+)))) . . . )
                    (st_2 (mult.behav-simul
                                n
                                (seq st
                                     (let ((st_1+ (system-process1-cycle
                                                       (system-update-signals st)))
                                          (st_2+ (mult.behav-simul
                                                     1
                                                     (seq st
                                                     < Affectation des ports d'entrée de mult.behav >
                                                     < Transmission des valeurs des ports de sortie > ))))
                                          (system-putst 'mult.behav st_2+ st_1+))
                                     < Affectation des ports d'entrée de mult.behav >
                                     < Transmission des valeurs des ports de sortie > ))))
                    (system-putst 'mult.behav st_2 st_1)))))
```

Ainsi, les propriétés sur la fonction `mult.behav-simul` peuvent s'exporter sur l'état global du système.

## 6.2 Utilisation de spécifications abstraites

L'idée, tirée de [79] , appliquée dans [84], consiste à utiliser le principe d'encapsulation d'Acl2 pour construire une spécification abstraite pour la simulation et la vérification.

Le principe d'encapsulation permet de déclarer de nouvelles fonctions, sans introduire leurs définitions, mais avec lesquelles nous associons un ensemble de théorèmes (les propriétés des fonctions). Une définition dite *locale* des fonctions (non visible à l'extérieur de la commande) permet l'acceptation des théorèmes: Ces définitions instancient les symboles de fonctions pour

admettre les théorèmes, puis disparaissent. La méthode d'application du principe d'encapsulation est la suivante : Considérons une architecture utilisant un composant, celui-ci ayant des propriétés (par exemple, effectuant une multiplication en 2 cycles de simulation). Le principe d'encapsulation permet de travailler sur les preuves de l'architecture englobante sans se soucier de la définition du composant. Ceci peut-être utile, non seulement pour pouvoir travailler sur des preuves de modèles VHDL en faisant l'hypothèse du bon fonctionnement des composants, divisant l'effort de preuves à plusieurs niveaux, mais également en caractérisant une famille de composants ayant les mêmes propriétés.

Le principe d'encapsulation apporte un autre avantage: puisqu'il existe au moins une fonction satisfaisant les propriétés à l'intérieur de l'encapsulation, celle-ci peut-être utilisée, à condition d'avoir une équivalence avec le modèle VHDL original, pour simuler le modèle VHDL entier. L'avantage étant de pouvoir apporter éventuellement un gain de temps pour la simulation, en remplaçant la définition du composant par un algorithme de haut niveau d'abstraction. Cette dernière caractéristique est importante vis à vis des tentatives d'industriels de spécifier des descriptions de haut niveau avec des langages tel que C ou C++. Ces langages sont populaires mais leur manque de sémantiques formelles les exclut de la vérification formelle.

Dans Acl2, les fonctions abstraites `component-simul`, `component-putst` et `component-getst` peuvent ainsi être introduites au moyen de la commande `encapsulate`. La figure 6.1 montre cette construction.
Les cinq premières lignes introduisent les fonctions abstraites avec leur arité: la fonction de simulation, les accesseurs, le prédicat reconnaisseur de l'état mémoire et un prédicat reconnaisseur de l'état initial.
Les cinq commandes qui suivent donnent les définitions "témoins" qui sont locales à la commande. Elles assurent qu'il existe au moins un composant satisfaisant les contraintes. Ces définitions sont imposées pour l'admissibilité de l'encapsulation par Acl2.

Les théorèmes qui suivent exposent la propriété exportable de ce modèle de composant:

**Théorème component-prod:** La valeur de sortie *prod* prend le résultat de la multiplication de $a$ avec $b$ après $a+2$ cycles de simulation. À ce moment la valeur de *done* est à 1 (True).

Le dernier théorème `component-simul-induction` dans l'`encapsulate` est un faux théorème requis par Acl2 pour générer une règle d'induction pour la fonction de simulation du système. Il indique que nous pouvons utiliser l'induction sur `component-simul` en utilisant le même schéma que `mult.behav-simul`. Ce théorème est nécessaire car la définition de la fonction de simulation est masquée, donc son schéma d'induction n'est pas visible à l'extérieur de l'encapsulation.

Cette facilité d'Acl2 permet des preuves hiérarchiques, l'abstraction d'un composant permet d'effectuer des preuves sur l'architecture englobante, preuves ainsi réalisées et valables pour tout composant pouvant s'instancier avec le modèle créé par `encapsulate`. En effet, en utilisant l'instanciation fonctionelle, l'encapsulation peut-être instanciée par une architecture réalisant le

```
(encapsulate
  (((component-simul * *) => *)
    ((component-getst *) => *)
    ((component-putst *) => *))
    ((component-statep *) => *)
    ((component-initstatep *) => *)
  (local (defun component-simul (n st) (mult.behav-simul n st)))
  (local (defun component-getst (var st) (mult.behav-getst var st)))
  (local (defun component-putst (var new st) (mult.behav-putst var new st)))
  (local (defun component-statep (st) (mult.behav-statep st)))
  (local (defun component-initstatep (st)
                (and (equal (mult.behav-getst 'mult-state st) 0)
                     (>= (mult.behav-getst 'a st) 1)
                     (>= (mult.behav-getst 'b st) 1))))

 ;; Propriétés associées aux signaux de sortie :
  (defthm component-prod
    (implies (and (true-listp st)
                  (component-statep st)
                  (component-getst 'req st)   ; signal de request
                  (component-initstatep st))
             (and
                (equal (component-getst 'prod
                                (component-simul
                                    (+ 2 (component-getst 'a st))
                                    st))
                   (* (component-getst 'a st)
                      (component-getst 'b st)))
                (equal (component-getst 'done
                                (component-simul
                                    (+ 2 (component-getst 'a st))
                                    st))
                     1))))

  (defthm component-simul-induction t
    :rule-classes ((:induction :pattern (component-simul n mem)
                               :condition t
                               :scheme (component-simul n mem)))))
```

FIG. 6.1 – *Encapsulation du composant mult.behav*

même calcul. Notamment l'instanciation d'une description de niveau algorithmique, plus rapide à l'exécution.

L'*instanciation fonctionnelle* permet d'apporter à la connaissance d'Acl2 (pour la preuve d'un théorème), des définitions aux fonctions abstraites. Ceci peut-être utile si, à un moment donné de la preuve, une ou plusieurs définitions précises sont nécessaires. L'instanciation vérifie que les définitions satisfont bien les propriétés de l'encapsulation initiale, mais les théorèmes supplémentaires que l'utilisateur aura prouvés *à partir de ceux présents dans l'encapsulation*, n'ont pas besoin d'être revérifiés. Sa syntaxe est la suivante :
`(:functional-instance lmi (f1 g1) ... (fn gn))` où lmi est théorème à instancier, chaque fi est une fonction "instanciable" d'arité ni de lmi, et gi est une fonction d'arité ni ou une expression lambda.

Considérons une architecture `arch` contenant un composant nommé `comp`. Le composant possède la même propriété que le composant `mult`, c'est-à-dire qu'elle réalise une multiplication des entrées (que nous nommons a et b) après a+2 cycles de simulation et que la valeur de done est à 1 à ce moment.

La construction suivante peut être ajoutée par l'utilisateur dans un théorème qu'il essaye de prouver sur comp-statep. Après vérification que le composant `comp` satisfait bien les propriétés présentes dans l'encapsulation, Acl2 va réutiliser le théorème
`theoreme_a_utiliser` (préalablement prouvé sur component-statep)
en instanciant `component-statep` par `comp-statep`.

```
..
:hints
  (("Goal" :by
     (:functional-instance theoreme_a_utiliser
        (component-statep comp-statep))))
```

Acl2 doit vérifier que la fonction introduite satisfait bien les contraintes de l'encapsulation, cette étape est la plus coûteuse en temps. Cependant nous garantissons que cette étape est raisonnablement facile, puisque cela dépend principalement de la structure des fonctions mises en oeuvre. Il est important de noter que les preuves des théorèmes prouvés entre l'encapsulation et l'instanciation fonctionnelle n'ont pas besoin d'être revérifiés. Il s'agit d'une grande économie de temps, permettant ainsi de développer des bibliothèques de théorèmes réutilisables sur des modèles de composants ou des modèles d'architectures (par exemple des preuves types d'architectures régulières).

# Chapitre 7

# Implémentation d'un prototype

Le but de l'implémentation d'un prototype est de rendre utilisable le modèle Acl2 d'une description VHDL. Une session Acl2 permet de définir et exécuter des fonctions, mais il n'est pas aisé :
- de redéfinir à chaque fois les fonctions et théorèmes Acl2 d'un modèle VHDL
- de manipuler l'état-mémoire.

Acl2 est basé sur une notion d'état globale, appelée *single-threaded state object*. Cet état n'est pas visible par l'utilisateur mais est utilisé pour gérer sa base de données, les définitions de fonctions et les théorèmes. `Ld` est la boucle d'évaluation des commandes Acl2. Elle gère l'affichage, évalue la commande de l'utilisateur et affiche le résultat sur l'écran. Notre contribution est d'écrire une nouvelle interface utilisateur adaptée à notre modèle, qui effectue de nouvelles commandes d'entrées/sorties et une nouvelle boucle d'évaluation. Ceci associé à une série de traducteurs partant du code source VHDL jusqu'à la formalisation dans Acl2. Les sources du projet sont sur [38].

## 7.1 Vue d'ensemble du fonctionnement

L'ensemble du prototype est schématisé sur la figure 7.1. L'interface utilisateur est le pivot gérant les traductions et pilotant Acl2 en fonction des choix de l'utilisateur.

Pour développer un traducteur transformant le source VHDL dans la logique d'Acl2 suivant la méthodologie présentée dans cette thèse, nous avons défini un format intermédiaire. La syntaxe de ce format, ainsi qu'un exemple, est présentée dans la section suivante.

Le principe de traduction d'un fichier source VHDL dans la logique d'Acl2 est le suivant :

- Traduction du source VHDL (fichier suffixé en .vhdl) vers le format intermédiaire (.env), appelons cette traduction T(vhdl->env). Ce traducteur est spécifié dans la section 7.3.
- Traduction du format intermédiaire vers les fonctions et théorèmes Acl2 correspondants : T(env->Acl2). Ce traducteur, présenté dans la section 7.3, est géré par une nouvelle interface utilisateur textuelle.

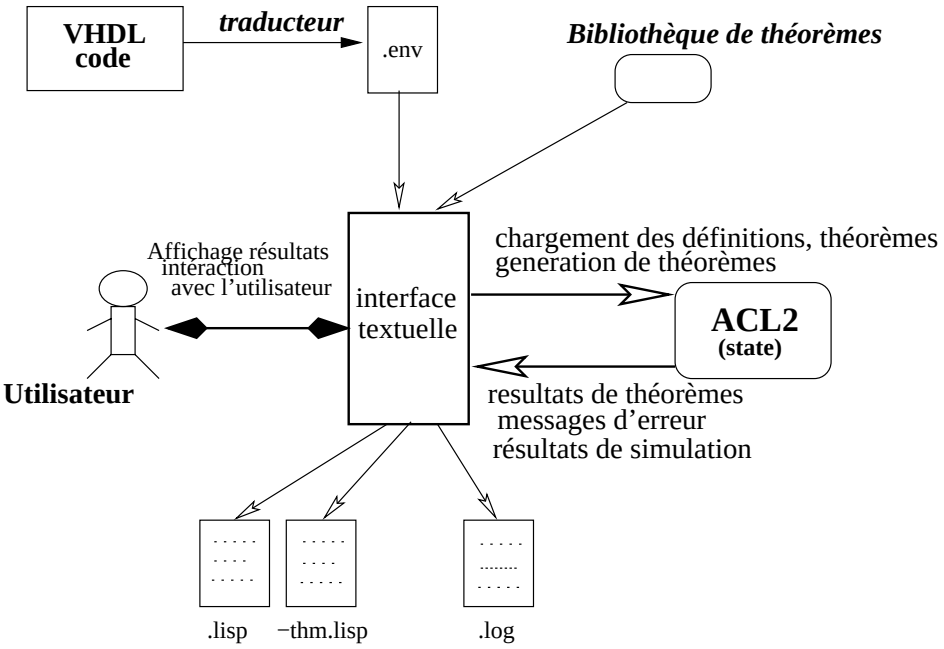


FIG. 7.1 – *Vue d'ensemble du prototype*

Les conventions de nommage de fichiers sont les suivantes :

- *arch-entity.vhdl* contient la description VHDL.
- *arch-entity.env* contient la représentation dans le format intermédiaire de la description VHDL.
- *arch-entity.lisp* contient toutes les fonctions Acl2 modélisant la description VHDL.
- *arch-entity-thm.lisp* contient les théorèmes Acl2 spécifiques à la description.
- *arch-entity.log* est le fichier accueillant la sortie standard d'Acl2, notamment les erreurs de simulation, d'ouverture de fichiers, etc ...

## 7.2 Définition du format intermédiaire .env

Le format intermédiaire est une sorte de "mise à plat" des caractéristiques de la description VHDL.

La forme intermédiaire est une *liste associative Common Lisp* :

Un mot-clé Common Lisp se présente sous la forme `:language item`, où `item` est la valeur correspondante au mot-clé `:language`. Ainsi, un modèle d'une description VHDL se présente sous la forme suivante :

(:mot-$cl_1$ $val_1$ :mot-$cl_2$ $val_2$ ... :mot-$cl_n$ $val_n$)

où les éléments $val_i$ représentent les valeurs des mot-clés.

Les descriptions VHDL modélisées dans ce format débutent par le mot-clé `:language VHDL`.

La description de chaque mot-clé est donnée pour un modèle VHDL. Les mots-clefs ne contenant pas de valeurs peuvent être omis du format. Par exemple, le mot-clé `:for-loop` peut être supprimé si la description VHDL ne contient pas de boucles for.

**:language :** Permet l'identification du langage source, cette identification est importante si le format est étendu à d'autres langages.

**:entity_name :** Nom de l'entité de la description.

**:inputs_signals :** Liste des noms des signaux d'entrées, il s'agit d'une liste d'identificateurs, par exemple (A B), l'ordre des déclarations peut être arbitraire.

**:inputs_type :** Liste des types correspondant aux signaux d'entrée. Cette liste doit respecter le même ordre que la liste donnée en argument de `:inputs_signals`. Il s'agit d'une liste d'identificateurs de types, par exemple (bit integer integer) (voir plus bas pour les identificateurs de types).

**:outputs_signals :** Liste des noms des signaux de sortie, il s'agit d'une liste d'identificateurs, par exemple (A B), l'ordre des déclarations peut être arbitraire.

**:outputs_type :** Liste des types correspondant aux signaux de sortie. Cette liste doit respecter le même ordre que la liste donnée en argument de `:outputs_signals`.

**:generic_parameters :** Liste des noms des paramètres génériques. Il s'agit d'une liste d'identificateurs, l'ordre des déclarations peut être arbitraire.

**:generic_parameters_type :** Liste des types correspondant aux paramètres génériques. Cette liste doit respecter le même ordre que la liste donnée en argument de `:generic_parameters`.

**:generic_parameters_default_value :** Liste des valeurs par défaut des paramètres génériques. Il s'agit d'une liste de constantes par exemple (1 4 3), l'ordre devant correspondre avec celui de la liste `:generic_parameters`.

**:architecture_name :** Nom de l'architecture de la description.

**:local_signals :** Liste des noms des signaux locaux, il s'agit d'une liste d'identificateurs, par exemple (A B), l'ordre des déclarations peut être arbitraire.

**:local_signals_type :** Liste des types correspondant aux signaux d'entrée. Cette liste doit respecter le même ordre que la liste donnée en argument de `:local_signals`.

**:local_variables :** Liste des noms des variables locales, il s'agit d'une liste d'identificateurs, par exemple (A B), l'ordre des déclarations peut être arbitraire.

**:local_variables_type :** Liste des types correspondant aux variables. Cette liste doit respecter le même ordre que la liste donnée en argument de `:local_variables`.

**:list_of_configuration :** Liste reliant le nom d'un composant avec le couple entité-architecture qui lui correspond. Il s'agit d'une liste de la forme

```
((component-name entity_name architecture_name)
 (component-name entity_name architecture_name)
 ...
)
```

**:list_of_components :** Liste des noms des exemplaires de composants, il s'agit d'une liste de mots-clés de la forme

```
((component-instanciation-label1 component-name)
 (component-instanciation-label2 component-name)
...
)
```

où component-instanciation-label est l'identificateur de l'exemplaire du composant et component-name l'identificateur du composant.

**:list_of_links :** Liste des correspondances des signaux d'interface du composant avec les signaux de l'architecture englobante. Il s'agit d'une liste de la forme :

```
(( (sig_comp1 sig_arch1)
   (sig_comp2 sig_arch2)
   ...
 )
 ...
)
```

**:list_for-statement :** Liste des instructions for...loop de la description. Chaque instruction for..loop étant caractérisée par une liste contenant les éléments suivants :

```
(ident direction instruction_bloc)
```

où :
- `ident` est un identificateur caractérisant la boucle
- `direction` est soit to, soit downto
- `instruction_bloc` est le bloc d'instructions (cf syntaxe des blocs d'instructions).

La boucle for..loop de la description doit ensuite être remplacée par l'appel à la fonction ident. En Lisp : (ident args).

**:list_functions :** Liste des fonctions de la descriptions. Chaque fonction étant caractérisée par la liste de mots-clés suivante :

```
(  :name ident
   :arg list_of_ident
   :type_args list_of_type_ident
   :type_return type_ident
   :local_variables list_of_ident
   :local_variables_type list_of_type_ident
   :body instruction_bloc
)
```

Cette liste fonctionne de la même manière que les autres listes de mots-clés. La valeur de retour de la fonction est spécifiée par la variable 'return, ainsi, une instruction (return <= exp) doit être présente dans la fonction.

**:list_process :** Liste des process de la description, chaque process étant caractérisé par la liste suivante :

```
(ident
 instructions_bloc
)
```

où ident est l'identificateur du process et instructions_bloc étant le bloc d'instructions (précisé dans la syntaxe).

**:concurrent-statement :** Liste des instructions concurrentes. Chaque instruction concurrente est modélisée par une liste d'affectations de signaux.

La syntaxe du format intermédiaire est donnée en annexe.

### 7.2.1 Exemple : la factorielle

Voici la description VHDL donnée en (figure 3.2), traduite dans le format intermédiaire :

```
(:language VHDL

:entity_name mysystem
:inputs_signals (arg start)
:inputs_type (natural bit)
:outputs_signals (res done)
:outputs_type (natural bit)
:architecture_name fact
:local_signals (op1 op2 resmult startmult endmult)
:local_signals_type (natural natural natural bit bit)
:local_variables (mystate r f)
:local_variables_type (natural natural natural)

:process
```

```
(
  (Mult
   (
    (if (= startmult 1)
      (
       (resmult <= (* op1 op2))
      )
    )
    (endmult <= startmult)
   )
  )

  (Doit
   (
    (if (= mystate 0)
      (
       (r := arg)
       (f := 1)
       (if (= start 1)
        (
         (mystate := 1)
        ))
      )
      (
       (if (= mystate 1)
        (
         (if (= r 1)
          (
           (res <= f)
           (done <= 1)
           (mystate := 0)
          )
          (
           (startmult <= 1)
           (op1 <= r)
           (op2 <= f)
           (mystate := 2)
          )
         )
        )
        (
         (if (= mystate 2)
          (
           (if (= endmult 1)
            (
             (f := resmult)
             (r := (- r 1))
             (startmult <= 0)
             (mystate := 1)
            )
           )
          )
         )
        )
       )
      )
    )
   )
  )
)
```

)

## 7.3 Le traducteur de VHDL vers les fichiers d'entrée d'Acl2

Le traducteur partant de VHDL est développé à partir d'un compilateur commercial: LVS de LEDA (Synopsys) par Pierre Ostier et Claude Le Faou. Il s'agit, à partir de la spécification donnée en annexe, d'écrire le traducteur partant de la représentation en arbre décoré de LVS pour aller vers le format intermédiaire.

### Generation des fichiers Acl2

Nous nous attardons ici sur le traducteur générant les fichiers Acl2 à partir d'un fichier .env.

Le principe est d'établir un modèle de fonctions à générer qui est ensuite *instancié* par la description à formaliser. L'instanciation est simplement l'écriture dans un fichier.

Plus précisément, nous établissons des "squelettes" de fonctions, par exemple, pour les accesseurs *getst* et *putst*, les squelettes sont donnés ci-dessous (Les éléments ~xi vont être remplacés par des éléments de la description à traiter). Ils suivent la méthodologie de définitions des accesseurs (voir section 3.5).

```
(defun ~x0 (var st)
       (declare (xargs :guard t)
                (type ~x3 var)
                (type (satisfies true-listp) st))
       (nth (~x1 var) st))

(defun ~x2 (var new st)
       (declare (xargs :guard t)
                (type ~x3 var)
                (type (satisfies true-listp) st))
       (update-nth (~x1 var) new st))
```

où,

**~x0 :** Il s'agit du nom *arch-entity*-getst.

**~x1 :** Il s'agit du nom *arch-entity*-getnth.

**~x2 :** Il s'agit du nom *arch-entity*-putst.

**~x3 :** Il s'agit de la liste totale des éléments de l'état-mémoire, précédée de `member` (cf section 3.5), par exemple :

```
(member arg
        start op1 op2 resmult startmult endmult
        op1+ op2+ resmult+ startmult+ endmult+
        mystate r f res done res+ done+)
```

La fonction qui génère les fonctions `getst` et `putst` se nomme `make-getst-putst`, elle prend en argument les listes des signaux d'entrées, des signaux locaux, des variables, des signaux de sorties et des noms de composants. Ces listes sont accessibles depuis le format intermédiaire (fichier `.env`) à l'aide de l'accesseur `cadr_assoc_keyword` qui donne la valeur du mot-clé donné en argument. L'appel de la fonction se fait comme suit (`list` représente le fichier `.env`) :

```
(make-getst-putst (cadr_assoc_keyword :entity_name list)
                  (cadr_assoc_keyword :architecture_name list)
                  (append (cadr_assoc_keyword :inputs_signals list)
                          (cadr_assoc_keyword :generic_parameters list))
                  (cadr_assoc_keyword :local_signals list)
                  (cadr_assoc_keyword :local_variables list)
                  (cadr_assoc_keyword :outputs_signals list)
                  (cadr_assoc_keyword :list_of_components list)
                  channel
                  state)
```

La définition de `make-getst-putst` est la suivante. Nous retrouvons dans la partie encadrée

les modèles des fonctions présentées en début de paragraphe.

```
(defun make-getst-putst (name_entity name_arch list_of_inputs list_of_locals_signals
                         list_of_variables list_of_outputs list_of_components
                         channel state)
(let ((total_list (append list_of_inputs list_of_locals_signals (make-next list_of_locals_signals)
                          list_of_variables list_of_components list_of_outputs
                          (make-next list_of_outputs))))
(fms "
;; Accessor of the memory state: (getst var st) -> value
(defun ~x0 (var st)
       (declare (xargs :guard t)
       (type ~x3 var)
       (type (satisfies true-listp) st))
(nth ( x1 var) st))
;; Updater of the memory state: (putst ver new st) -> st
(defun  x2 (var new st)
       (declare (xargs :guard t)
                (type ~x3 var)
                (type (satisfies true-listp) st))
       (update-nth (~x1 var) new st))

;;;;;;;;;;;;;;;;"
(list
(cons #\0 (make-name name_entity name_arch "-getst"))
(cons #\1 (make-name name_entity name_arch "-get-nth"))
(cons #\2 (make-name name_entity name_arch "-putst"))
(cons #\3 (append (list 'member) total_list))
)
channel
state
nil)))
```

– *Explication*:

Nous employons la fonction d'entrées/sorties `fms` d'Acl2. La fonction `fms` est un substitut de la fonction *format* en Common Lisp, c'est-à-dire une fonction d'affichage sur un canal de sortie (écran ou fichier). La syntaxe de `fms` est la suivante : *(fms string alist channel state evisc-tuple)*, où

- `string` est la chaîne de caractères à afficher, cette chaîne peut contenir des paramètres (noté ~xi), dont les valeurs sont contenues dans le paramètre `alist`.
- `alist` contient les valeurs des paramètres xi. `alist` est de la forme (`#\0` signifie le caractère 0):

```
(list
      (cons #\0 ; valeur du paramètre ~x0 )
      (cons #\1 ; valeur du paramètre ~x1)
... )
```

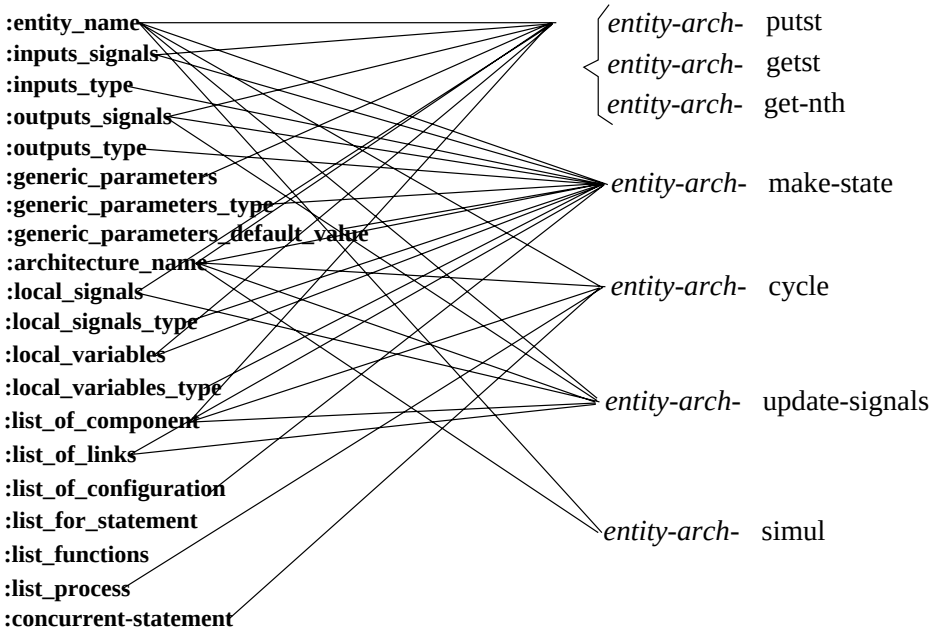


FIG. 7.2 – *Liens mot-clés - fonctions Acl2*

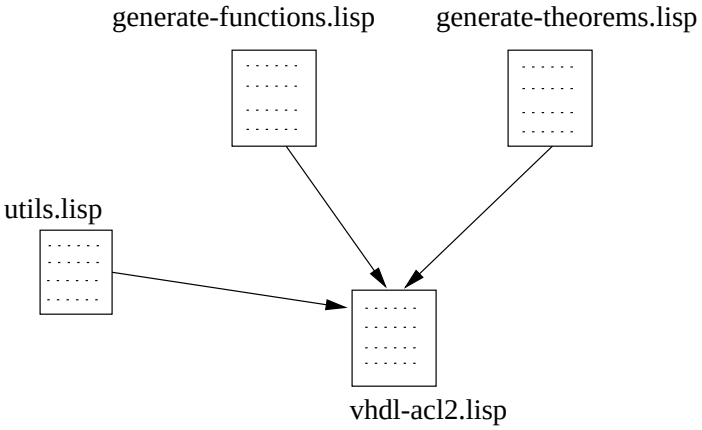


FIG. 7.3 – *Structures des fichiers*

- `channel` est le canal de sortie (*standard-co* est la sortie sur écran, sinon le résultat de l'appel: (open-output-channel string :object state) avec *string* étant le nom d'un fichier).
- `state` est l'état globale d'Acl2.
- `evisc-tuple` est à `nil` pour notre usage.

Dans la définition de `make-getst-putst`, la fonction `make-next`, appliquée à `list_of_outputs` ou `list_of_local_signals`, crée une liste pour les valeurs futures des signaux. Par exemple
`(make-next '(s done res))` rend `(s+ done+ res+)`. La fonction `make-name` crée un nouveau symbole en concatenant ses arguments.

– *Fin explication*

Les autres fonctions modélisant la description VHDL sont écrits sous le même principe. La figure 7.2 montre les liens entre les éléments du format intermédiaire et les fonctions générées.

La hiérarchie des fichiers source du prototype est présentée sur la figure 7.3.

## 7.4 Exemple de session

Pour démarrer une session, il est nécessaire de charger le fichier principal (`(ld ``vhdl-acl2.lisp'')`), puis de démarrer par `(init state)`. Une session l'outil débute ainsi :

```
ACL2 !>(ld ``vhdl-acl2.lisp'')
 ...
ACL2 !>(init state)
```

– *Sortie de l'outil*

```
vhdl-Acl2 1.0 (c) P. Georgelin
-------------------------------------
            =============
          ||  MAIN MENU  ||
            =============

  1 - Choose an environment file
  2 - Load a description

q:to quit|h:help
-------------------------------------
Vhdl_Acl2>
```

– *Fin de sortie* où l'interface propose deux choix d'actions à l'utilisateur.

- La première appele le traducteur, qui traduit un fichier `.env` en fichier `.lisp` (contenant la formalisation en Acl2) et en `-thm.lisp` contenant les théorèmes sur le modèle.
- Le second choix propose de charger une description. L'interface charge les fichiers `.lisp` et `-thm.lisp` correspondants.

Lorsqu'un modèle Acl2 d'une description VHDL est chargé, la structure des menus est la suivante :

**1 - Numeric simulation :** L'outil affiche chaque signal d'entrée, ainsi que son type, l'utilisateur fournit une valeur pour chacun. Il fournit ensuite le nombre de cycle de simulation. L'outil effectue l'évaluation et affiche le résultat.

**2 - Symbolic simulation :** L'outil affiche chaque signal d'entrée ou local à l'architecture. L'utilisateur a le choix de rentrer une valeur numérique ou symbolique. L'outil fournit le résultat de simulation symbolique.

**3 - Properties :** L'outil demande le nombre de cycles de simulation souhaité et la propriété désirée.

Voici un exemple d'une simulation numérique, puis de deux simulations symboliques.

Convention de notations :

- Les éléments en **gras** représentent l'entrée de l'utilisateur
- Les éléments écrits en type `courier` représentent la sortie d'Acl2.
- Les éléments en *italique* sont des commentaires.

```
   ACL2 !>(init state)
vhdl-Acl2  P. Georgelin (07-2001)
-----------------------------------
             =============
           ||  MAIN MENU  ||
             =============

  1 - Choose an environment file
  2 - Load a description

q:to quit|h:help
-----------------------------------
```

`Vhdl_Acl2>` **1** – *Choix 1 : Traducteur*

`Choose name (terminated by ") :` "**fact**"

– *Sortie de l'outil*

```
Generating function get-nth ...
Generating predicate state-p ...
Generating macro make-state ...
Generating function update memory state ...
Generating one function per process ...
Generating function update signals ...
Generating function for one simulation cycle ...
Generating function for N simulation cycle ...

 -------  File "fact.lisp" generated --------

generating lemma for updater
generating lemma : cycle function is well-formed
------  File "fact-thm.lisp" generated -------
```

– *fin de sortie*

```
vhdl-Acl2  P. Georgelin (07-2001)
-----------------------------------
             =============
           ||  MAIN MENU  ||
             =============
```

```
  1 - Choose an environment file
  2 - Load a description
q:to quit|h:help
-----------------------------------
Vhdl_Acl2> 2 – Choix 2: chargement d'une description

Choose name (terminated by ") : "fact"
Opening fact...

-----------------------------------
          Description Menu
-----------------------------------

   1 - Numeric simulation
   2 - Symbolic simulation
            --
   3 - Properties
q:return |h:help
-----------------------------------
loaded : FACT
Vhdl_Acl2> 1 – Choix 1 : Simulation numérique
Choose value for input : ARG (NATURAL) : "7"
Choose value for input : START (BIT) : "1"
Choose value for input : OP1 (NATURAL) : "0"
Choose value for input : OP2 (NATURAL) : "0"
Choose value for input : RESMULT (NATURAL) : "0"
Choose value for input : STARTMULT (BIT) : "0"
Choose value for input : ENDMULT (BIT) : "0"
  How many simulation cycle : "23"
– Sortie de l'outil
ACL2 Version 2.5.  Level 2.  Cbd "/home/preuves/georgelin/src/vhdl-acl2/".
Type :help for help.
ACL2 !>>
  ARG             : 7
  START           : 1
  OP1             : 7
  OP2             : 1
  RESMULT         : 5040
  STARTMULT       : 1
  ENDMULT         : 0
  OP1+            : 7
  OP2+            : 1
```

```
  RESMULT+        : 7
  STARTMULT+      : 1
  ENDMULT+        : 1
  MYSTATE         : 2
  R               : 7
  F               : 1
  RES             : 5040
  DONE            : 1
  RES+            : 5040
  DONE+           : 1
ACL2 !>>Bye.
```

*– fin sortie*

```
------------------------------------
          Description Menu
------------------------------------

   1 - Numeric simulation
   2 - Symbolic simulation
              --
   3 - Properties


q:return |h:help
------------------------------------
loaded : FACT
Vhdl_Acl2> 2 – Choix 2: Simulation symbolique
Choose value for input : ARG (NATURAL) : "q"
Choose value for input : START (BIT) : "1"
Choose value for input : OP1 (NATURAL) : "op1"
Choose value for input : OP2 (NATURAL) : "op2"
Choose value for input : RESMULT (NATURAL) : "resmult"
Choose value for input : STARTMULT (BIT) : "startmult"
Choose value for input : ENDMULT (BIT) : "endmult"
Choose value for input : OP1+ (NATURAL) : "op1"
Choose value for input : OP2+ (NATURAL) : "op2"
Choose value for input : RESMULT+ (NATURAL) : "0"
Choose value for input : STARTMULT+ (BIT) : "0"
Choose value for input : ENDMULT+ (BIT) : "1"
Choose value for input : MYSTATE (NATURAL) : "2"
Choose value for input : R (NATURAL) : "R"
Choose value for input : F (NATURAL) : "F"
```

```
How many simulation cycle ? : "1"
```
*– Sortie de l'outil*
```
  OP1           : OP1
  OP2           : OP2
  RESMULT       : 0
  STARTMULT     : 0
  ENDMULT       : 1
  STARTMULT+    : 0
  ENDMULT+      : 0
  MYSTATE       : 1
  R             : (+ -1 r)
  F             : 0
  OP1+          : RESMULT+
  OP2+          : STARTMULT+
```
*– Fin sortie*
```
================================================
                      MENU
================================================

   1 - Numeric simulation
   2 - Symbolic simulation
             --
   3 - Properties

q:return |h:help
================================================
loaded : FACT
Vhdl_Acl2> 2
```
*– Choix 2 : Simulation symbolique*
```
Choose value for input : ARG (NATURAL) : "q"
Choose value for input : START (BIT) : "1"
Choose value for input : OP1 (NATURAL) : "op1"
Choose value for input : OP2 (NATURAL) : "op2"
Choose value for input : RESMULT (NATURAL) : "resmult"
Choose value for input : STARTMULT (BIT) : "startmult"
Choose value for input : ENDMULT (BIT) : "endmult"
Choose value for input : OP1+ (NATURAL) : "op1+"
Choose value for input : OP2+ (NATURAL) : "op2+"
Choose value for input : RESMULT+ (NATURAL) : "resmult"
Choose value for input : STARTMULT+ (BIT) : "1"
Choose value for input : ENDMULT+ (BIT) : "0"
Choose value for input : MYSTATE (NATURAL) : "1"
Choose value for input : R (NATURAL) : "5"
```

```
Choose value for input : F (NATURAL) : "F"

How many simulation cycle ? : "1"
```
*– Sortie de l'outil*
```
  OP1           : OP1+
  OP2           : OP2+
  RESMULT       : RESMULT
  STARTMULT     : 1
  ENDMULT       : 0
  OP1+          : 5
  OP2+          : F
  RESMULT+      : (* op1+ op2+)
  STARTMULT+    : 1
  ENDMULT+      : 1
  MYSTATE       : 2
  OP1+          : RESMULT+
  OP2+          : STARTMULT+
```
*– Fin sortie*
```
================================================
                      MENU
================================================

   1 - Numeric simulation
   2 - Symbolic simulation
             --
   3 - Properties

q:return |h:help
================================================
loaded : FACT
Vhdl_Acl2> 2
```
*– Choix 2: Simulation symbolique*
```
Choose value for input : ARG (NATURAL) : "q"
Choose value for input : START (BIT) : "1"
Choose value for input : OP1 (NATURAL) : "op1"
Choose value for input : OP2 (NATURAL) : "op2"
Choose value for input : RESMULT (NATURAL) : "resmult"
Choose value for input : STARTMULT (BIT) : "startmult"
Choose value for input : ENDMULT (BIT) : "endmult"
Choose value for input : OP1+ (NATURAL) : "op1+"
Choose value for input : OP2+ (NATURAL) : "op2+"
Choose value for input : RESMULT+ (NATURAL) : "resmult+"
Choose value for input : STARTMULT+ (BIT) : "1"
```

```
Choose value for input : ENDMULT+ (BIT) : "0"
Choose value for input : MYSTATE (NATURAL) : "1"
Choose value for input : R (NATURAL) : "r"
Choose value for input : F (NATURAL) : "f"

How many simulation cycle ? : "1"
```

*– Sortie de l'outil*

```
IF (equal r 1) then
  OP1           : OP1+
  OP2           : OP2+
  RESMULT       : RESMULT+
  STARTMULT     : 1
  ENDMULT       : 0
  RESMULT+      : (* op1+ op2+)
  ENDMULT+      : 1
  MYSTATE       : 0
  OP1+          : RESMULT+
  OP2+          : STARTMULT+
  RESMULT+      : F
  STARTMULT+    : 1
Else
  OP1           : OP1+
  OP2           : OP2+
  RESMULT       : RESMULT+
  STARTMULT     : 1
  ENDMULT       : 0
  OP1+          : R
  OP2+          : F
  RESMULT+      : (* op1+ op2+)
  STARTMULT+    : 1
  ENDMULT+      : 1
  MYSTATE       : 2
  OP1+          : RESMULT+
  OP2+          : STARTMULT+
```

*– fin de sortie*

Exemple de preuves de propriétés:

```
================================================
                    MENU
================================================
```

```
   1 - Numeric simulation
   2 - Symbolic simulation
             --
   3 - Properties

q:return |h:help
================================================
loaded : FACT
Vhdl_Acl2> 3

================================================
               PROPERTY MENU
================================================
   1 - Property
   2 - Inductive property
             --
q:return |h:help
================================================
loaded : "FACTORIAL"
Vhdl_Acl2> 1
Choose value for input : ARG (natural) : "q"
Choose value for input : START (bit) : "1"
How many simulation cycle : "12"
Choose hypothesis :"(integerp q) (> q 4)"
Choose a name of local or output signal : "resmult"
Property :"(* q (- q 1) (- q 2) (- q 3))"
```

*– Sortie de l'outil*

```
ACL2 !>>
:-) Congratulations. Your property is True
ACL2 !>>Bye.
```

*– Fin de sortie*

# Chapitre 8

# Conclusion et Perspectives

## 8.1 Notre contribution

Les outils de vérification formelle doivent permettre aux concepteurs de vérifier des descriptions complexes et de pouvoir raisonner sur des ensembles de valeurs très grands voire infinis. Les outils actuels tels que les Model-checkers sont restrictifs car ils ne peuvent travailler sur des niveaux d'abstraction plus hauts que le niveau Transfert de Registres (RTL) ou des modèles similaires en machines d'états finis. De plus, ils sont limités sur le nombre total d'états et ne fournissent pas de support pour raisonner sur des architectures régulières, paramétrées ou compositionnelles. Les démonstrateurs de théorèmes ne souffrent pas de ces limitations mais ne possèdent pas d'automatismes et de méthodologies permettant leur utilisation dans une chaîne de vérification.

Notre méthode a été influencée par l'analyse de circuits de taille réelle qui nous ont été fournis dans ce but par STMicroelectronics. Les circuits analysées sont des circuits de haut niveau d'abstraction, dédiés au traitement du signal (codage, décodage) [77, 42]. Cependant, en raison de la complexité des opérations traitées (décodage Viterbi, Transformée de Fourier, ..), il nous a été seulement possible de faire des simulations numériques et symboliques ainsi que des preuves sur ces résultats. L'étude de ces descriptions, ainsi que de celles réalisants une factorielle ou une multiplication par additions itérées, à plusieurs process ou plusieurs composants, ont permis de :

- fournir une méthode de formalisation de la sémantique d'un sous-ensemble VHDL vers la logique d'Acl2.
- développer un traducteur VHDL vers démonstrateur.
- fournir un moteur avec interface utilisateur pour manipuler ce modèle: simulation numérique à une vitesse comparable à C, et simulation symbolique. Cette interface génère également des fonctions et théorèmes nécessaires de façon automatique.
- développer des méthodes de preuves raisonnant par cycles de simulation, dont des méthodes de preuves inductives.

Ce sous-ensemble VHDL ne contient pas de caractéristiques spécifiques par rapport aux diverses définitions sémantiques formelles [35, 14, 66]), il est cependant assez grand pour travailler sur les types de circuits qui nous ont été fournis et permet d'écrire des traducteurs formalisant ce sous-ensemble VHDL dans Acl2.

La spécification et la réalisation de traducteurs, ainsi que d'une interface Acl2 spécifique aux modèles VHDL, permettent un gain de temps sensible en formalisation et preuves.

Nous pensons que cette contribution fournit une voie vers l'automatisation progressive de preuves par démonstration de théorèmes, les rendant accessibles aux ingénieurs de vérification. Les spécialistes de cette technique peuvent se libérer de certaines tâches de formalisation et de preuves, diminuant ainsi l'étape de vérification, de plus en plus longue pour une conception donnée. Les opérations effectuées (simulations numériques et symboliques, preuves) se basent sur un même modèle, renforçant ainsi la confiance que peut avoir le concepteur sur la formalisation. Nous sommes convaincus que le spécialiste ne pourra être remplacé pour les preuves de théorèmes complexes, longues et non-automatisables, qui sont généralement propres à une seule description.

Des travaux actuels tentent de "marier" des techniques de vérifications formelles, notamment Model-Checking (ou techniques de manipulation de graphe) et démonstration de théorèmes [5, 55, 63, 76, 70, 81]. afin de profiter des avantages de chacun et en éliminant leur inconvénients. Il en ressort plusieurs méthodes comme développer des procédures de déduction dans un démonstrateur pour implémenter l'efficacité des modèles-checkers symboliques, ou alors développer des techniques d'abstractions et de déductions sur des model-checkers. Notre contribution tente de marier la technique de simulation symbolique et la démonstration de théorèmes.
En effet, dans notre cas, nous utilisons les techniques de simplifications et d'abstractions d'Acl2 pour réduire les résultats de simulations symboliques.

## 8.2 Perspectives

Les perspectives de ce travail sont nombreuses, elles peuvent se porter tout d'abord à court terme sur trois axes majeurs :

- Augmentation du sous-ensemble VHDL traité : notamment en permettant les instructions **for generate** ou les descriptions asynchrones. Un début de travail sur les process combinatoires à été fait.
- Augmentation des bibliothèques de théorèmes réutilisables et augmentation des méthodes de preuves pour valider des architectures régulières de composants [31], notamment en utilisant les résultats sur des types d'architectures formalisées avec Nqthm dans [68].
- Élaboration de types de données, d'opérateurs et de théorèmes pour les opérations de traitement du signal pour la preuve de circuits DSP. Pouvoir prouver que l'implémentation d'un produit de convolution ou d'une transformée de Fourier est correcte est un véritable défi actuellement.

Une comparaison, dans [80, 70], entre le Model-Checking et la démonstration de théorème, est faite pour la vérification de l'opération de transformée inverse du cosinus. Les auteurs utilisent le Model-checker SMV et le théorème "Chinese Remainder" pour réduire les termes. Il en résulte que cette technique possède comme avantage de prouver facilement des propriétés sur les dépassements de capacités. Un Model-checker travaillant au niveau des vecteurs de bits prend en compte (par la taille de ces vecteurs) la précision limite de l'implémentation des nombres réels. La technique de démonstrateur de théorème est ici plus efficace car utilisant une abstraction plus élevée. Un ensemble d'axiomes sur les nombres réels et la transformée inverse du cosinus permettent d'avoir une spécification algorithmique, à comparer à l'implémentation. L'analyse bit par bit des vecteurs n'est, en général, pas nécessaire pour la preuve. Il est néanmoins important de disposer d'une bonne implémentation des nombres réels avec des indications de précisions limites.

Pour réellement prouver le bon comportement de ce type de circuit, il est nécessaire de dévelop-

per une bibliothèque Acl2 des opérations de traitement du signal. Cette bibliothèque, prouvée correcte, pouvant ensuite servir de spécifications. David Russinoff a développé une bibliothèque sur les opérateurs arithmétiques basée sur les nombres flottants [73], il s'agit comme perspective, pour ces types de circuits, de compléter cette bibliothèque.

- Preuves sur les résultats de la synthèse de haut niveau : Julia Dushina [12, 36] propose une méthodologie de vérification fondée sur un modèle de machine abstraite. Un prototype d'outil de vérification, utilisant le système Prolog et basée sur la simulation symbolique, extrait les opérations effectuées par l'architecture finale et les compare ensuite avec les opérations spécifiées dans la machine abstraite correspondant au circuit avant l'allocation. Le "mariage" de cet outil avec un démonstrateur de théorème permettra de prouver l'équivalence des opérations sémantiquements (et non seulement syntaxiquement) égales.

Une des perspectives les plus prometteuses à plus long terme est l'adaptation de nouveaux langages, notamment des langages tels que SpecC [37], ou Java, permettant ainsi des preuves d'architectures mixtes matériels-logiciels réutilisables.

# Annexe A

# Exemple de session Acl2

Convention de notations :

- Les éléments en **gras** représentent l'entrée de l'utilisateur
- Les éléments écrits en type `courier` représentent la sortie d'Acl2.
- Les éléments en *italique* sont des commentaires.

*Soumission d'une définition de fonction dans Acl2 :*

```
ACL2 !>(defun app (x y)
           (cond ((endp x) y)
                 (t (cons (car x)
                          (app (cdr x) y)))))
```

*– Sortie du démonstrateur :*

```
The admission of APP is trivial, using the relation E0-ORD-< (which
is known to be well-founded on the domain recognized by E0-ORDINALP)
and the measure (ACL2-COUNT X).  We observe that the type of APP is
described by the theorem (OR (CONSP (APP X Y)) (EQUAL (APP X Y) Y)).
We used primitive type reasoning.


Summary
Form:  ( DEFUN APP ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.03 seconds (prove: 0.00, print: 0.00, other: 0.03)
 APP
```

*– Fin sortie*

*Exemples d'exécution de la fonction app introduite :*

```
ACL2 !>(app nil '(x y z))
(X Y Z)
```

ACL2 !>**(app '(1 2 3) '(4 5 6 7))**
(1 2 3 4 5 6 7)

ACL2 !>**(app '(a b c d e f g) '(x y z))**
(A B C D E F G X Y Z)

ACL2 !>**(app (app '(1 2) '(3 4)) '(5 6))**
(1 2 3 4 5 6)

ACL2 !>**(app '(1 2) (app '(3 4) '(5 6)))**
(1 2 3 4 5 6)

*Soumission d'un théorème à Acl2 :*

```
ACL2 !>(defthm associativity-of-app
                (equal (app (app a b) c)
                       (app a (app b c))))
```

*– Sortie du démonstrateur :*

```
Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes are
suggested by this conjecture.  Subsumption reduces that number to two.
However, one of these is flawed and so we are left with one viable
candidate.


We will induct according to a scheme suggested by (APP A B).  If we
let  (:P A B C) denote *1 above then the induction scheme we'll use
is
(AND
   (IMPLIES (AND (NOT (ENDP A))
                 (:P (CDR A) B C))
            (:P A B C))
   (IMPLIES (ENDP A) (:P A B C))).
This induction is justified by the same argument used to admit APP,
namely, the measure (ACL2-COUNT A) is decreasing according to the relation
E0-ORD-< (which is known to be well-founded on the domain recognized
by E0-ORDINALP).  When applied to the goal at hand the above induction
scheme produces the following two nontautological subgoals.

Subgoal *1/2
(IMPLIES (AND (NOT (ENDP A))
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C))))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).
```

```
By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/2'
(IMPLIES (AND (CONSP A)
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C))))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).


But simplification reduces this to T, using the :definition APP, the
:rewrite rules CDR-CONS and CAR-CONS and primitive type reasoning.

Subgoal *1/1
(IMPLIES (ENDP A)
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).


By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP A))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

But simplification reduces this to T, using the :definition APP and
primitive type reasoning.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM ASSOCIATIVITY-OF-APP ...)
Rules: ((:REWRITE CDR-CONS)
        (:REWRITE CAR-CONS)
        (:DEFINITION NOT)
        (:DEFINITION ENDP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:DEFINITION APP))
Warnings:  None
Time:  0.27 seconds (prove: 0.10, print: 0.05, other: 0.12)
 ASSOCIATIVITY-OF-APP
```

– *Fin sortie*

# Annexe B

# Les classes de règles d'Acl2

Lorsque l'utilisateur prouve un théorème, celui-ci, par défaut, devient une nouvelle règle de réécriture. Cependant, l'utilisateur a la possibilité de préciser le type de règle qu'il va générer en prouvant un théorème. Nous énumérons ici les principales classes de règles disponibles et nous présentons ensuite en détail quelques unes des règles (réécriture, induction) utilisées dans le cadre de notre travail.

La syntaxe d'utilisation est la suivante :

(**defthm** *nom_théorème*
(implies (hyps x y z) *Partie hypothèses*
(concl x y z)) *Partie conclusion*
:rule-classes (:REWRITE :GENERALIZE ...)) *Indication du type de règle désiré*

## B.1 Les classes de règles :

**Built-in-clauses :** Pour construire une clause dans le simplificateur.

**Compound-recognizer :** Permet d'ajouter une règle utilisée par le mécanisme de typage.

**Congruence definition :** Permet de définir les relations à maintenir au moment de la simplifications d'arguments.

**Elim :** Permet de définir une règle d'élimination de destructeurs.

**Equivalence :** Permet de caractériser une relation comme étant une relation d'équivalence.

**Forward-chaining :** Permet de définir des règles de chaînage avant lorsqu'une certaine hypothèse se produit.

**Generalize :** Permet de définir des règles de généralisations.

**Induction :** Permet de définir des règles qui suggèrent une certaine induction.

**Linear :** Permet de définir des règles d'inégalité arithmétique.

**Meta :** Permet de définir une règle :meta (un simplificateur manuel).

**Refinement :** Permet de préciser qu'une relation d'équivalence affine une autre.

**Rewrite :** Permet de définir des règles de réécritures.

**type-prescription :** Permet de définir une règle qui spécifie le typage d'un terme.

**Well-founded-relation :** Montre qu'une relation est bien fondée sur un ensemble.

## B.2 La classe :rewrite

La classe :rewrite permet de définir des règles de réécritures.
Par exemple:
`(equal (+ x y) (+ y x))` remplace (+ a b) par (+ b a) (pourvu que certaines heuristiques approuvent la permutation).

```
(implies (true-listp x)              remplace (append a nil) par a, si
         (equal (append x nil) x)) (true-listp a) se réécrit en t.
 (implies
     (and (eqlablep e)        remplace (member a (append b c)) par
          (true-listp x)      (member a (append c b)) dans les contextes où
          (true-listp y)      les hypothèses (eqlablep e)
     (iff (member e (append x y))     (true-listp x) et (true-listp c)
          (member e (append y x))))  se réécrivent en t.
```

Forme générale :

```
(and ...
     (implies (and ...hi...)
              (implies (and ...hk...)
                       (and ...
                            (equiv lhs rhs)
                            ...)))
     ...)
```

Une règle de réécriture est utilisée lorsque toutes instances du terme `lhs` (left-hand side) apparaît dans un contexte où la relation d'équivalence est effective. En premier lieu, Acl2 tente de trouver une substitution entre le terme `lhs` et le terme cible. Ensuite, il tente de vérifier les hypothèses instanciées de la règle.

Si les hypothèses sont vérifiées, ainsi que certaines restrictions qui évitent des boucles infinies, la cible est alors remplacée par le terme instancié `rhs` (right-hand side).

## B.3 La classe :induction

La classe :rewrite permet de définir des règles qui suggèrent une certaine induction.

Exemple:

```
(:induction :corollary t  ; le théorème prouvé n'est pas utile pour
            cet exemple
            :pattern (* 1/2 i)
            :condition (and (integerp i) (>= i 0))
            :scheme (recursion-by-sub2 i))
```

Dans Acl2, les fonctions dans une conjecture "suggèrent" les inductions utilisées par le système. En effet, toutes les fonctions récursives doivent être admises avec une justification en termes de mesure qui décroit de façon bien-fondée sur un ensemble donné d'arguments. Ainsi, les fonctions récursives suggèrent

un double schéma d'induction qui "déplie" les fonctions à partir d'une application donnée.

Par exemple, `append` décompose son premier argument en appliquant l'opérateur `cdr` aussi longtemps que la liste n'est pas `nil`, le schéma d'induction suggéré par `(append x y)` possède :
- un cas de base supposant x comme étant soit `nil`, ou d'un autre type que `true-listp`,
- un pas d'induction dans lequel l'hypothèse d'induction est obtenue en remplaçant `x` par `(cdr x)`. Cette substitution diminue la même mesure utilisée pour justifier la définition de `append`.

Cependant, Acl2, contrairement à Nqthm, fournie un moyen par lequel l'utilisateur peut construire des règles d'induction. Il s'agit de la classe de règle `:induction`. Le principe de définition crée automatiquement une règle `:induction`, nommée `(induction fn)`, pour chaque fonction admise `fn`. Il s'agit de la règle qui relie les applications de `fn` au schéma d'induction qu'il suggère.

La commande `defthm` est, en quelque sorte "surchargée" pour accueillir la possibilité de créer sa règle de type `:induction`, en effet, aucun théorème n'a besoin d'être prouvé pour réaliser le lien heuristique représentée par une règle `:induction`.

Forme générale pour un champ lemma ou Corollary dans un théorème :induction:
T

Forme générale d'un champ :rule-classes :induction :

```
(:induction :pattern pat-term
            :condition cond-term
            :scheme scheme-term)
```

où :
- `pat-term, cond-term, scheme-term` sont tous des termes, `pat-term` est l'application de la fonction `fn`, `scheme-term` est l'application de la fonction `rec-fn`, qui suggère une induction. Enfin, toutes les variables libres présentes dans `cond-term` et `scheme-term` sont des variabels libres de `pat-term`.

La règle d'induction crée est utilisée comme suit : Lorsque :
- une instance du terme `:pattern` apparaît dans une conjecture qui doit être prouvée par induction,
- l'instance correpondante du terme `:condition` est connue comme étant T,
l'instance correspondante du terme `:scheme` est créée et la règle "suggère" l'induction donnée par ce terme. Si `rec-fn` est récursive, alors la suggestion est l'une qui "déplie" cette récursion. Par exemple, analysons l'exemple donné ci-dessus :

```
(:induction :pattern (* 1/2 i)
            :condition (and (integerp i) (>= i 0))
            :scheme (recursion-by-sub2 i)).
```

Dans cet exemple, supposons que recursion-by-sub2 est la fonction :

```
(defun recursion-by-sub2 (i)
  (if (and (integerp i)            ; si i  est un entier > 1
           (< 1 i))
      (recursion-by-sub2 (- i 2))  ; récursion sur i-2
      t))                          ; sinon rend t.
```

Nous ne préoccupons pas ici de la valeur de la fonction, mais seulement de son schéma d'induction :

```
(and (implies (not (and (integerp i) (< 1 i)))   ; cas de base
              (:p i))
     (implies (and (and (integerp i) (< 1 i))    ; pas d'induction
                   (:p (- i 2)))
              (:p i)))
```

Supposons que la règle `:induction` a été ajoutée. Appellons ce schéma d'induction, une induction sur "i par 2".

Alors une occurence, par exemple, de `(* 1/2 k)` dans une conjecture qui doit être prouvée par induction, suggérerait, via cette règle, une induction de k par 2, si k est un entier positif. le terme `:pattern` apparaissant dans la conjecture, sa condition étant satisfaite, et le schéma étant suggéré par `(recursion-by-sub2 k)`.

# Annexe C

# Simulation symbolique de la description Mult

Sortie du simulateur symbolique de la description Mult donnée en figure 5.4.

```
(cond ((equal (mult-getst 'mult-state st) 0) ;;; Cas où la variable mystate=0
       (if (equal (mult-getst 'req st) 1)
          (mult-putst 'mult-state 1
                      (mult-putst 'count
                                  (mult-getst 'a st)
                                  (mult-putst 'prod 0
                                              (mult-putst 'c
                                                          (mult-getst 'c+ st)
                                              (mult-putst 'done
                                                          (mult-getst 'done+ st)
                                              st)))))
          (mult-putst 'c
                      (mult-getst 'c+ st)
                      (mult-putst 'done
                                  (mult-getst 'done+ st)
                                  st))))
       ((equal (mult-getst 'mult-state st) 1) ;;; Cas où la variable mystate=1
       (cond ((equal (mult-getst 'req st) 0) ;;; Si req = 0
              (mult-putst 'mult-state 0
                          (mult-putst 'c
                                      (mult-getst 'c+ st)
                                      (mult-putst 'done
                                                  (mult-getst 'done+ st)
                                                  st))))
              ((< 0 (mult-getst 'count st))
              (mult-putst 'count (+ -1 (mult-getst 'count st))
                          (mult-putst 'prod
                                      (+ (mult-getst 'b st)
                                         (mult-getst 'prod st))
                                      (mult-putst 'c
```

```
                                    (mult-getst 'c+ st)
                                    (mult-putst 'done
                                                (mult-getst 'done+ st)
                                                st)))))
        (t
        (mult-putst 'mult-state 2
                    (mult-putst 'done+
                                1
                                (mult-putst 'c+
                                            (mult-getst 'prod st)
                                            (mult-putst 'c
                                                        (mult-getst 'c+ st)
                                                        (mult-putst 'done
                                                                    (mult-getst 'done+ st)
                                                                    st)))))))))
((equal (mult-getst 'mult-state st) 2) ;;; Cas où mystate=2
(if (equal (mult-getst 'req st) 0) ;;; Si req = 0
   (mult-putst 'mult-state 0
               (mult-putst 'done+
                           0
                           (mult-putst 'c+
                                       0
                                       (mult-putst 'c
                                                   (mult-getst 'c+ st)
                                                   (mult-putst 'done
                                                               (mult-getst 'done+ st)
                                                               st)))))
   (mult-putst 'c
               (mult-getst 'c+ st)
               (mult-putst 'done
                           (mult-getst 'done+ st)
                           st))))
(t (mult-putst 'mult-state
               0
               (mult-putst 'c
                           (mult-getst 'c+ st)
                           (mult-putst 'done
                                       (mult-getst 'done+ st)
                                       st)))))
```

# Annexe D

# Syntaxe du format intermédiaire

Notations :
- Les mots en minuscule, certains contenant le caractère '_', sont utilisés pour dénoter les catégories syntaxiques. Par exemple, list_of_components
- Les mots en gras ou utilisant le caractère ':' sont utilisés pour dénoter des mots réservés, par exemple
:language, to, ...
- Une barre verticale représente le "ou"
- Les termes entre crochets sont opionnels
- Les termes entre accolades sont des ensembles de ces termes. Le terme peut apparaître 0 ou plus de fois. La répétition se fait de la gauche vers la droite.

VHDL model ::=
(
language
entity_name
[io]
[generic]
architecture_name
[locals]
[components]
[list_for-statement]
[list_functions]
[list_process]
[conc-assignment]
)

io ::=
[input_signals input_type]
[output_signals output_types]

generic ::=
generic_parameters

generic_parameters_type
generic_parameters_default_value

locals ::=
[local_signals local_signals_type]
[local_variables local_variables]

componentss ::=
list_of_component
list_of_links
list_of_configuration

language ::=
:language VHDL

entity_name ::=
:entity_name ident

inputs_signals ::=
:inputs_signals list_of_ident

inputs_type ::=
:inputs_type list_of_type_ident

outputs_signals ::=
:outputs_signals list_of_ident

outputs_type ::=
:outputs_type list_of_type_ident

generic_parameters ::=
:generic_parameters list_of_ident

generic_parameters_type ::=
:generic_parameters_type list_of_type_ident

generic_parameters_default_value ::=
:generic_parameters_default_value list_of_constant_number

architecture_name ::=
:architecture_name ident

local_signals ::=
:local_signals list_of_ident

local_signals_type ::=
:local_signals_type list_of_type_ident

```
local_variables ::=
:local_variables list_of_ident

local_variables_type ::=
:local_variables_type list_of_type_ident

list_of_components ::=
:list_of_components list_of_comp_decl

list_of_links ::=
:list_of_links list_of_comp_inst

list_of_configuration ::=
:list_of_configuration list_of_comp_conf

list_for-statement ::=
:for-statement
( for-statement )

list_functions ::=
:list_of_functions
(  functions  )

list_process ::=
:process
( Process  )

concurrent-statement ::=
:concurrent-statement
signal_assignment

Process ::=
(ident instructions_bloc)

Ident ::= any char or word ; e .g desc-arch, rec_idt_fmt, ...

Function ::=
(               :name ident
                :arg list_of_ident
                :type_args list_of_type_ident
                :type_return type_ident
                :local_variables list_of_ident
                :local_variables_type list_of_type_ident
                :body instruction_bloc
)
```

Liste_of_ident ::=
( ident)

Liste_of_type_ident ::=
( type_ident)

Constant number ::= any integer ; e.g 4, 16, É

Variables assignment ::=
(target := expr)

Signal assignment ::=
(target ¡= expr)

Target ::=
Ident [(constant_number [direction constant_number])]

Expr ::=
Ident [(constant_number [direction constant_number])]
| constant_number
| function_call

IF statement ::=
(**if** conditional_stat
    ( Instruction_bloc)
    ( Instruction_bloc)
)

Conditional_stat ::=
(**equal** Expr Expr)

Function call ::= (function_name args)

Args ::=
 Expr

list_of_comp_conf ::=
(  comp_conf  )

comp_conf ::=
(name_instance_of_component
entity_name_of_component
architecture_name_of_component)
Type identifier ::=
bit | std_logic | integer | positive | natural | signed | unsigned —
(**bit_vector** constant_number) | (**std_logic_vector** constant_number)

Direction ::=
**to** | **downto**

Instruction_bloc ::=
( variables assignment | Signal assignment | IF statement |)

FOR statement ::=
(ident direction instruction_bloc)

List_of_comp_decl ::=
( comp_decl )

Comp_decl ::=
(:component-instanciation-label component-name)

List_of_comp_inst ::=
( comp_inst )

Comp_inst ::=
list_of_links_for_component-instanciation-label

list_of_links_for_component-instanciation-label ::=
( correspondance )

Correspondance ::=
:signal_ident_of_component signal_ident_of_architecture

# Annexe E

# Spécification du traducteur VHDL vers le format intermédiaire

Convention de notations :
- Les éléments en **gras** représentent l'entrée de l'utilisateur
- Les éléments écrits en type `courier` représentent les unités syntaxiques.
- Les éléments en *italique* sont des commentaires.

## E.1 Syntaxe VHDL du sous-ensemble traité

*Déclaration de l'entité:*

entity_declaration ::= **entity** identifier **is**
entity_header
**end [entity_simple_name]**;
entity_header ::= [generic_clause] [port_clause]
generic_clause ::= **generic** (generic_list);
generic_list ::= generic_interface_list
port_clause ::= **port** (port_list) ;
port_list ::= port_interface_list
generic_interface_list ::= { generic_interface_element; }
generic_interface_element ::=
identifier_list ::= subtype_indication [:= static_expression]
port_interface_list ::= { port_interface_element; }
port_interface_element ::= [interface_in_declaration]
[interface_out_declaration]
interface_in_declaration ::=
[identifier_list : **in** subtype_indication [:= static_expression] ]
interface_in_declaration ::=
[identifier_list : **out** subtype_indication [:= static_expression] ]

*Déclaration de l'architecture:*

architecture_body ::=
**architecture** identifier **of** entity_name bfseries is

architecture_declarative_part
**begin**
architecture_statement_part
**end** [architecture_name];
architecture_declarative_part ::= { bloc_declarative_item }
architecture_statement_part ::= {concurrent_statement}
bloc_declarative_item ::=
subprogram_declaration
| subprogram_body
| signal_declaration
| configuration_specification
subprogram_body ::= subprogram_specification **is**
subprogram_declarative_part
**begin**
subprogram_statement_part
**end** [designator]
subprogram_specification ::=
**function** designator [(formal_parameter_list)] **return** type_mark
subprogram_declarative_part ::= { subprogram_declarative_item
subprogram_statement_part ::= { sequential_statement }
subprogram_declarative_item ::= variable_declaration
formal_parameter_list ::= *parameter*_interface_list
*parameter*_interface_list ::= *parameter*_interface_element { ; *parameter*_interface_element }
*parameter*_interface_element ::= identifier_list : subtype_indication
sequential_statement ::= signal_assignment_statement
| variable_assignment_statement
| if_statement
| loop_statement
| return_statement
variable_assignment_statement ::= target := expression
signal_assignment_statement ::= target ¡= expression
target ::= name
if_statement ::= **if** condition **then**
sequence_of_statements_*then*
[ **else** sequence_of_statements_*else* ]
**end if;**
loop_statement ::= [loop_label: ] iteration_scheme **loop**
sequence_of_statements
**end loop** [loop_label];
iteration_scheme ::= **for** *loop*_parameter_specification
*loop*_parameter_specification ::= identifier **in** discrete_range
discrete_range ::= integer1 direction integer2
direction ::= to | downto
return_statement = **return** [expression];
configuration_specification ::= **for** component_specification **use**
concurrent_statement ::= process_statement
component_instanciation_statement

concurrent_signal_assignment_statement

process_statement ::= [process_label:]
**process**
process_declarative_part
**begin**
process_statement_part
**end process** [process_label];

process_declarative_part ::= { process_declarative_item }

process_declarative_item ::= variable_declaration

process_statement_part ::= {sequential_statement}

sequential_statement ::= signal_assignment_statement
| variable_assignment_statement
|if_statement
|loop_statement

component_instanciation_statement ::= instanciation_label
component_name
[generic_map_aspect]
[port_map_aspect];

generic_map_aspect ::= **generic map** (generic_association_list)

port_map_aspect ::= **port map** (port_association_list)

association_list_element ::= name1 => name2

association_list ::= (association_list_element {,association_list_element})

concurrent_signal_assignment_statement ::= target <= expression;

## E.2 Traduction

Convention de notations :

- Les éléments en **gras** représentent les mots-clés du format intermédiaire.
- Les fonctions présentes ici (T, T2, T2-comp1....)sont les fonctions de traductions, certaines sont récursives. T est la fonction principale.
- Les éléments en *italique* sont des commentaires.
- le symbole @ est le symbole de concaténation.

*Spécification de l'entité:*

T(entity_declaration) = **:entity** identifier
T(entity_header)
T(entity_header) = [T (generic_clause)]
[T (port_clause)]
T(generic_clause) = T(generic_list)
T(generic_list) = T-rec1(*generic*_interface_list)
T-rec1(*generic*_interface_list) = T-rec1par(*generic*_interface_list)
T-rec1sub(*generic*_interface_list)
T-rec1def(*generic*_interface_list)
T-rec1par(generic_interface_list) = **:generic_parameters**
@(T-rec2(generic_interface_element)
T-rec1par(generic_interface_list))
T-rec1arg(generic_interface_list) = **:generic_parameters_type**
@(T-rec1argsub(generic_interface_element)
T-rec1arg(generic_interface_list))
T-rec1def(generic_interface_list) = **:generic_parameters_default_values**
@(T1-rec1defstat(generic_interface_element)
T-rec1def(generic_interface_list))
T-rec1argsub(generic_interface_element) = T1(subtype_indication longueur(identifier_list))
T-rec1argstat = (static_expression longueur(identifier_list))

T-rec2(generic_interface_element) = T-rec2(identifier_list)
T-rec2(identifier_list) = ( *identifier* T-rec2(identifier_list) )
T1(indication L) =
T-rec3(indication L)
T-rec3(indication L) =
( *indication* T-rec3(subtype_indication (L-1)))

T(port_clause) = T(port_list)
T(port_list) = T(port_interface_list)
T(port_interface_list) = [T2-rec ( { interface_in_declaration } )]
[T2-rec ( { interface_out_declaration } )]

*Traduction des declarations des ports d'entrées:*

T2-rec( {interface_in_declaration}) =
**:input_signals** T2-rec_ident({interface_in_declaration})
**:input_signals_type** T2-rec_subtype({interface_in_declaration})

**:input_signals_default_values** T2-rec_static({interface_in_declaration})
*Traduction des identificateurs :*

T2-rec_ident({interface_in_declaration}) =
@(T2-rec_ident1(interface_in_declaration)
T2-rec_ident({interface_in_declaration}))
T2-rec_ident1(interface_in_declaration) = T-rec2(identifier_list)
*Identificateurs de types des ports d'entrées :*

T2-rec_subtype({interface_in_declaration}) =
@(T2-rec_subtype1(interface_in_declaration)
T2-rec_subtype({interface_in_declaration}))
T2-rec_subtype1(interface_in_declaration) = T2(subtype_indication longueur(identifier_list))
*Traduction des valeurs par défaut des ports d'entrées :*

T2-rec_static({interface_in_declaration}) =
@(T2-rec_static1(interface_in_declaration)
T2-rec_static({interface_in_declaration}))
T2-rec_static1(interface_in_declaration) = T1(static_expression longueur(identifier_list))
*Traduction des déclarations des ports de sorties :*

T2-rec( {interface_out_declaration}) =
**:output_signals** T2-rec_ident({interface_out_declaration})
**:output_signals_type** T2-rec_subtype({interface_out_declaration})
*Identificateurs des ports de sortie :*

T2-rec_ident({interface_out_declaration}) =
@(T2-rec_ident1(interface_out_declaration)
T2-rec_ident({interface_out_declaration}))
T2-rec_ident1(interface_out_declaration) = T-rec2(identifier_list)
*Identificateurs de types des ports de sorties :*

T2-rec_subtype({interface_out_declaration}) =
@(T2-rec_subtype1(interface_out_declaration)
T2-rec_subtype({interface_out_declaration}))
T2-rec_subtype1(interface_out_declaration) = T2(subtype_indication longueur(identifier_list))

*Spécification de l'architecture:*

T(architecture_body) = **:architecture_name** identifier
T(architecture_declarative_part)
T(architecture_statement_part)
T(architecture_declarative_part) = T( block_declarative_item )
T(block_declarative_item) = **:list_of_functions** T2( {subprogram_body })
**:local_signals** T2( {signal_declaration })
**:list_of_component** T2-comp1( {configuration_specification })
**:list_of_configuration** T2-comp2( {configuration_specification })

```
T2-comp1( {configuration_specification}) =
T2( {subprogram_body }) = @(T2sub(subprogram_body)
                            T2( {subprogram_body }))
T2sub(subprogram_body) = (T2sub1(subprogram_specification)
                          T2sub2(subprogram_declarative_part)
                          :body T2sub3(suprogram_statement_part)
                          )

T2sub1(subprogram_specification) = :name designator
                                   T2subarg(formal_parameter_list)
                                   :type_return T2subreturn(type_mark)
T2subarg(formal_parameter_list) = T2subargident(formal_parameter_list)
                                  T2subargtype(formal_parameter_list)
T2subargident(formal_parameter_list) = :arg @(T-rec2(interface_element)
                                              T2subargident(formal_parameter_list))
T2subargtype(formal_parameter_list) = :type_arg @(T-rec1argsub(interface_element)
                                                  T2subargtype(formal_parameter_list))

T2sub2(suprogram_declarative_part) = T2sub2varident({variable_declaration })
                                     T2sub2vartype({variable_declaration })
T2sub2varident({variable_declaration }) = :local_variables
                                          @(T-rec2(interface_element)
                                            T2sub2varident({variable_declaration }))
T2sub2vartype({variable_declaration }) = :local_variables_type
                                         @(T-rec1argsub(interface_element)
                                           T2sub2vartype({variable_declaration }))

T2sub3(subprogram_statement_part) = T-seq({sequential_statement })

T-seq({sequential_statement }) = (T-seq1(sequential_statement)
                                  T-seq({sequential_statement })
                                  )
T-seq1(sequential_statement) = T-seq1sig(signal_assignment_statement)
                               |T-seq1var(variable_assignment_statement)
                               |T-seq1if(if_statement)
                               |T-seq1loop(loop_statement)
                               |T-seq1ret(return_statement)

T-seq1sig(signal_assignment_statement) = (target <= expression)
T-seq1var(variable_assignment_statement) = (target := expression)
T-seq1if(if_statement) = (if condition
                          T-seq(sequence_of_statements_then)
                          T-seq(sequence_of_statements_else)
                          )
T-seq1loop(loop_statement) = (loop_label integer1 integer2)
T-seq1ret(return_statement) = (return := expression)
```

T(architecture_statement_part) = T({concurrent_statement})
T({concurrent_statement}) = **:list_process** T({process_statement})
**:list_of_links** T({component_instanciation_statement})
**:concurrent-statement** T({concurrent_signal_assignment_statement})
T({process_statement}) = (@(Tp(process_statement)
T({process_statement})))
Tp(process_statement) = Tp(process_statement_part)
Tp(process_statement_part) = T-seq({sequential_statement })
T({component_instanciation_statement}) = @(Tc(component_instanciation_statement)
T({component_instanciation_statement}))
Tc(component_instanciation_statement) = (component_name (@(Tc1(generic_map) Tc2(port_map))))
Tc1(generic_map) = Tc1(association_list)
Tc2(port_map) = Tc1(association_list)
Tc1(association_list) = @(Tc-rec(association_list_element)
Tc1(association_list))
Tc-rec(association_list_element) = (name1 name2)
T({concurrent_signal_assignment_statement}) = (@(Ta(concurrent_signal_assignment_statement)
T({concurrent_signal_assignment_statement})))

*Pour les variables locales :*

LV(architecture) = LV(architecture_statement_part)
LV(architecture_statement_part) = **:local_variables** LV-ident({process_statement})
**:local_variables_type** LV-type({process_statement})
LV-ident({process_statement}) = @(LV-ident1(process_statement)
LV-ident({process_statement}))
LV-ident1(process_statement) = LV-ident1(process_declarative_part)
LV-ident1(process_declarative_part) = LV-ident1(variable_declaration)
LV-ident1(variable_declaration) = @(T2-rec(interface_element)
LV-ident1(variable_declaration))

LV-type({process_statement}) = @(LV-type1(process_statement)
LV-type({process_statement}))
LV-type1(process_statement) = LV-type1(process_declarative_part)
LV-type1(process_declarative_part) = LV-type1(variable_declaration)
LV-type1(variable_declaration) = @(T-rec1argsub(interface_element)
LV-type1(variable_declaration))

# Annexe F

# Scripts Acl2

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; File "fact.lisp"
;;
;; generated by vhdl-Acl2 v.1.0 07/2001
;; P. Georgelin
;; email : Philippe.Georgelin@imag.fr
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(in-package "ACL2")

(include-book "/home/preuves/georgelin/src/vhdl-acl2/utils")

(include-book "/home/preuves/georgelin/src/vhdl-acl2/expander")

;;; Function get-nth : gives the position of an element
;;; in the memory state

(defun FACT-GET-NTH (var)
   (declare (type
(member arg
        start op1 op2 resmult startmult endmult
        op1+ op2+ resmult+ startmult+ endmult+
        mystate r f res done res+ done+)
var)
    (xargs :guard t))
   (the (integer 0 18)

(case var ('arg 0)
      ('start 1)
      ('op1 2)
      ('op2 3)
      ('resmult 4)
      ('startmult 5)
      ('endmult 6)
      ('op1+ 7)
      ('op2+ 8)
      ('resmult+ 9)
      ('startmult+ 10)
      ('endmult+ 11)
      ('mystate 12)
      ('r 13)
      ('f 14)
      ('res 15)
      ('done 16)
      ('res+ 17)
      ('done+ 18)))
```

```
)


;; Accessor of the memory state : (getst var st) -> value

(defun FACT-GETST (var st)
  (declare (xargs :guard t)
   (type
(member arg
        start op1 op2 resmult startmult endmult
        op1+ op2+ resmult+ startmult+ endmult+
        mystate r f res done res+ done+)
var)
           (type (satisfies true-listp) st))
  (nth (FACT-GET-NTH var) st))


;; Updater of the memory state : (putst ver new st) -> st

(defun FACT-PUTST (var new st)
  (declare (xargs :guard t)
   (type
(member arg
        start op1 op2 resmult startmult endmult
        op1+ op2+ resmult+ startmult+ endmult+
        mystate r f res done res+ done+)
var)
           (type (satisfies true-listp) st))
  (update-nth (FACT-GET-NTH var) new st))


;;;;;;;;;;;;;;


(defun FACT-STP (st)
 (declare (xargs :guard t))

(and (true-listp st)
     (= (length st) 19)
     (naturalp (FACT-getst 'arg st))
     (bitp (FACT-getst 'start st))
     (naturalp (FACT-getst 'op1 st))
     (naturalp (FACT-getst 'op2 st))
     (naturalp (FACT-getst 'resmult st))
     (bitp (FACT-getst 'startmult st))
     (bitp (FACT-getst 'endmult st))
     (naturalp (FACT-getst 'op1+ st))
```

```
     (naturalp (FACT-getst 'op2+ st))
     (naturalp (FACT-getst 'resmult+ st))
     (bitp (FACT-getst 'startmult+ st))
     (bitp (FACT-getst 'endmult+ st))
     (naturalp (FACT-getst 'mystate st))
     (naturalp (FACT-getst 'r st))
     (naturalp (FACT-getst 'f st))
     (naturalp (FACT-getst 'res st))
     (bitp (FACT-getst 'done st))
     (naturalp (FACT-getst 'res+ st))
     (bitp (FACT-getst 'done+ st))))



;;;;;;;  This macro make-state creates an initial state ;;;;;

(defmacro FACT-MAKE-STATE
(&key (arg '0)
      (start '0)
      (op1 '0)
      (op2 '0)
      (resmult '0)
      (startmult '0)
      (endmult '0)
      (op1+ '0)
      (op2+ '0)
      (resmult+ '0)
      (startmult+ '0)
      (endmult+ '0)
      (mystate '0)
      (r '0)
      (f '0)
      (res '0)
      (done '0)
      (res+ '0)
      (done+ '0))
(list 'quote
(list arg
      start op1 op2 resmult startmult endmult
      op1+ op2+ resmult+ startmult+ endmult+
      mystate r f res done res+ done+))
)


;;;;;  This function updates memory state ;;;;
```

```
(defun FACT-UPDATE-ST (key-arg st)
  (update-state-body "MYSYSTEM" "FACT" key-arg st))

;; This function represents the process MULT ;;;;;;;

(defun FACT-MULT-CYCLE (st)
(seq st
     (if (= (FACT-getst 'startmult st) 1)
         (seq st
              (FACT-putst 'resmult+
                                  (* (FACT-getst 'op1 st)
                                     (FACT-getst 'op2 st))
                                  st))
         st)
     (FACT-putst 'endmult+
                         (FACT-getst 'startmult st)
                         st)))


;; This function represents the process DOIT ;;;;;;;

(defun FACT-DOIT-CYCLE (st)
(seq
 st
 (if
  (= (FACT-getst 'mystate st) 0)
  (seq st
       (FACT-putst 'r
                           (FACT-getst 'arg st)
                           st)
       (FACT-putst 'f 1 st)
       (if (= (FACT-getst 'start st) 1)
           (seq st (FACT-putst 'mystate 1 st))
           st))
  (seq
   st
   (if
    (= (FACT-getst 'mystate st) 1)
    (seq st
         (if (= (FACT-getst 'r st) 1)
             (seq st
                  (FACT-putst 'res+
                                      (FACT-getst 'f st)
                                      st)
                  (FACT-putst 'done+ 1 st)
                  (FACT-putst 'mystate 0 st))
             (seq st (FACT-putst 'startmult+ 1 st)
```

```
                (FACT-putst 'op1+
                                    (FACT-getst 'r st)
                                    st)
                (FACT-putst 'op2+
                                    (FACT-getst 'f st)
                                    st)
                (FACT-putst 'mystate 2 st))))
    (seq
       st
       (if (= (FACT-getst 'mystate st) 2)
           (seq st
                (if (= (FACT-getst 'endmult st) 1)
                    (seq st
                         (FACT-putst 'f
                                             (FACT-getst 'resmult st)
                                             st)
                         (FACT-putst 'r
                                             (- (FACT-getst 'r st) 1)
                                             st)
                         (FACT-putst 'startmult+ 0 st)
                         (FACT-putst 'mystate 1 st))
                  st))
           st)))))))



;;======== Function update of signals  =====


(defun FACT-UPDATE-SIGNALS (st)
(seq st
     (FACT-putst 'op1
                         (FACT-getst 'op1+ st)
                         st)
     (FACT-putst 'op2
                         (FACT-getst 'op2+ st)
                         st)
     (FACT-putst 'resmult
                         (FACT-getst 'resmult+ st)
                         st)
     (FACT-putst 'startmult
                         (FACT-getst 'startmult+ st)
                         st)
     (FACT-putst 'endmult
                         (FACT-getst 'endmult+ st)
                         st)
     (FACT-putst 'res
```

```
                        (FACT-getst 'res+ st)
                        st)
     (FACT-putst 'done
                        (FACT-getst 'done+ st)
                        st)))



;;======== Function cycle of simulation  =====


(defun FACT-CYCLE (st)
 (seq st (FACT-mult-cycle st) (FACT-doit-cycle st)))




;;======== Function for N simulation cycles  =====


(defun FACT-SIMUL (n st)
   (if (zp n)
      st

(FACT-simul (1- n)
                    (FACT-cycle (FACT-update-signals st))))
)



:comp t

; *** END OF FILE ***
```

```
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;        proofs.lisp
;;;
;;; This file contains proofs for the validation of fact.lisp
;;; August 17, 1999
;;; The path to the arithmetic book has to be adjusted according
;;; to each installation
;;; Philippe Georgelin
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(in-package "ACL2")

;(include-book "/projects/acl2/v2-4/books/arithmetic/top")
(include-book "/home/preuves/georgelin/src/acl2-sources/books/arithmetic/top")

(include-book "oldfact")

(defthm update-nth_consp
  (implies (consp st)
   (consp (update-nth n new_value st))))

(defthm length_update-nth
  (equal (len (update-nth n new_value st))
 (len st)))

;;; These two theorems allow to decompose update-nth in later proofs

(defthm lemma_nth_update-nth1
  (implies (and (integerp n) (<= 0 n) (consp st) (< n (len st)))
   (equal (nth n (update-nth n val st))
  val)))

(defthm lemma_nth_update-nth2
  (implies (and (integerp n) (<= 0 n)
(< n (len st))
(not (equal n m)))
   (equal (nth n (update-nth m val st))
  (nth n st))))

; ----------------------------------------
;;; Basic properties on fact.doit-cycle
; ----------------------------------------

(defthm len_fact.doit-cycle
  (equal (len (fact.doit-cycle st))
 (len st)))
```

```
; ----------------------------------------
;   State properties about fact.doit-cycle
; ----------------------------------------

(defthm fact.doit-cycle_not_modified
  (implies (and (equal (len st) 20)
(member i '(0 1 5 6 14 16)))
   (equal (nth i (fact.doit-cycle st))
  (nth i st))))

(defthm startmult_not_modified1
  (implies (and (equal (len st) 20)
(equal (nth 17 st) 1)
(equal (nth 8 st) 0)
(not (equal (nth 18 st) 1)))
   (not (equal (nth 8 (fact.doit-cycle st))
       1))))

;;;  result+ takes the value of doit.f or not after one cycle of fact.doit-cycle

(defthm lemma_fact.doit-cycle1
  (implies (equal (len st) 20)
   (equal (nth 10 (fact.doit-cycle st))
  (if (equal (nth 17 st) 1)
      (if (equal (nth 18 st) 1)
  (nth 19 st)
(nth 10 st))
    (nth 10 st)))))

;;; The value of 'done+ is equal to 1 after one cycle of fact.doit-cycle
;;; if doit.r is equal to 1

(defthm lemma_fact.doit-cycle2
  (implies (equal (len st) 20)
   (equal (nth 11 (fact.doit-cycle st))
  (if (equal (nth 17  st) 1)
      (if (equal (nth 18 st) 1)
  1
(nth 11  st))
    (nth 11 st)))))

;;; The value of 'op1+ is equal to 'doit.r if doit.r is not equal
;;; to 1 and 'doit.mystate is equal to 1

(defthm lemma_fact.doit-cycle3
  (implies (equal (len st) 20)
```

```
   (equal (nth 12 (fact.doit-cycle st))
  (if (equal (nth 17 st) 1)
      (if (equal (nth 18 st) 1)
  (nth 12 st)
(nth 18 st))
    (nth 12 st)))))

;;; The value of 'op2+ is equal to 'doit.f if doit.r is not equal
;;; to 1 and 'doit.mystate is equal to 1

(defthm lemma_fact.doit-cycle4
  (implies (equal (len st) 20)
   (equal (nth 13 (fact.doit-cycle st))
  (if (equal (nth 17 st) 1)
      (if (equal (nth 18 st) 1)
  (nth 13 st)
(nth 19 st))
    (nth 13 st)))))

;;; The value of 'startmult+ can be equal to 1 or 0 after one cycle
;;; of fact.doit-cycle

(defthm len-consp
  (implies (< 0 (len st))
   (consp st)))

(defthm lemma_fact.doit-cycle5
  (implies (equal (len st) 20)
   (equal (nth 15 (fact.doit-cycle st))
  (if (equal (nth 17 st) 1)
      (if (equal (nth 18 st) 1)
  (nth 15 st)
1)
    (if (equal (nth 17 st) 2)
(if (equal (nth 9 st) 1)
    0
  (nth 15 st))
      (nth 15 st))))))

;;; The value of 'doit.r can be equal to (1- doit.r) or
;;; arg

(defthm lemma_fact.doit-cycle6
  (implies (equal (len st) 20)
   (equal (nth 18 (fact.doit-cycle st))
  (if (equal (nth 17 st) 2)
      (if (equal (nth 9 st) 1)
```

```
  (1- (nth 18 st))
(nth 18 st))
    (if (equal (nth 17 st) 0)
(nth 0 st)
      (nth 18 st))))))

;;; The value of doir.f can be equal to resmult or 1 after one cycle
;;; of fact.doit-cycle

(defthm lemma_fact.doit-cycle7
  (implies (equal (len st) 20)
   (equal (nth 19 (fact.doit-cycle st))
  (if (equal (nth 17 st) 2)
      (if (equal (nth 9  st) 1)
  (nth 7 st)
(nth 19 st))
    (if (equal (nth 17 st) 0)
1
      (nth 19 st))))))

;;; Lemma about new values of doit.mystate after one cycle
;;; of fact.doit-cycle

(defthm lemma_fact.doit-cycle8
  (implies (equal (len st) 20)
   (equal (nth 17 (fact.doit-cycle st))
  (if (equal (nth 17 st) 0)
      (if (equal (nth 1 st) 1) 1 0)
    (if (equal (nth 17 st) 1)
(if (equal (nth 18 st) 1) 0 2)
      (if (equal (nth 17 st) 2)
  (if (equal (nth 9  st) 1)
      1 2)
(nth 17 st)))))))

; -----------------------------------------
;;; Basic properties of fact.multiplier-cycle
; -----------------------------------------

(defthm len_fact.multiplier-cycle
  (equal (len (fact.multiplier-cycle st))
 (len st)))

; -----------------------------------------
;   State properties about fact.multiplier-cycle
; -----------------------------------------
```

```
(defthm fact-multiplier-cycle_not_modified
  (implies (and (equal (len st) 20)
(member i '(0 1 10 11 12 13 15 17 18 19)))
   (equal (nth i (fact.multiplier-cycle st))
  (nth i st))))

(defthm endmult+_takes_startmult
  (implies (equal (len st) 20)
   (equal (nth 16 (fact.multiplier-cycle st))
  (nth 8 st))))

;;; resmult+ takes the result of (* op1 op2)

(defthm lemma_fact.multiplier-cycle1
   (implies (equal (len st) 20)
            (equal (nth 14 (fact.multiplier-cycle st))
                   (if (equal (nth 8 st) 1)
                       (* (nth 5 st) (nth 6 st))
                       (nth 14 st)))))

; -----------------------------------------
;;; Basic properties about fact-cycle
; -----------------------------------------

(defthm length_fact-cycle_lemma
              (equal (len (fact-cycle st))
                     (len st))
     :hints (("Goal"
            :in-theory (disable fact.multiplier-cycle fact.doit-cycle))))

; -----------------------------------------
;   State properties about fact-cycle
; -----------------------------------------

(defthm start_not_modified3
  (implies (and (equal (len st) 20)
(member i '(0 1)))
   (equal (nth i (fact-cycle st))
  (nth i st)))
  :hints (("Goal"
   :in-theory (disable fact.doit-cycle fact.multiplier-cycle))))

;;; This lemma introduces rewriting rules on the updating of signals

(defthm lemma_update-signals
  (implies (equal (len st) 20)
   (and (equal (nth 3 (fact-cycle st))
```

```
       (nth 10 (fact-cycle st)))
(equal (nth 4 (fact-cycle st))
       (nth 11 (fact-cycle st)))
(equal (nth 5 (fact-cycle st))
       (nth 12 (fact-cycle st)))
(equal (nth 6 (fact-cycle st))
       (nth 13 (fact-cycle st)))
(equal (nth 7 (fact-cycle st))
       (nth 14 (fact-cycle st)))
(equal (nth 8 (fact-cycle st))
       (nth 15 (fact-cycle st)))
(equal (nth 9 (fact-cycle st))
       (nth 16 (fact-cycle st)))))
  :hints (("Goal"
   :in-theory (disable fact.doit-cycle fact.multiplier-cycle))))

;;; The value of op1+ after one cycle of simulation

(defthm lemma_fact-cycle3
   (implies (equal (len st) 20)
            (equal (nth 12 (fact-cycle st))
                   (if (equal (nth 17 st) 1)
                       (if (equal (nth 18 st) 1)
                           (nth 12 st)
                           (nth 18 st))
                       (nth 12 st))))
 :hints (("Goal"
           :in-theory (disable fact.doit-cycle fact.multiplier-cycle))))

;;; The value of op2+ after one cycle of simulation

(defthm lemma_fact-cycle4
   (implies (equal (len st) 20)
            (equal (nth 13 (fact-cycle st))
                   (if (equal (nth 17 st) 1)
                       (if (equal (nth 18 st) 1)
                           (nth 13 st)
                           (nth 19 st))
                       (nth 13 st))))
:hints (("Goal"
           :in-theory (disable fact.doit-cycle fact.multiplier-cycle))))

;;; The value of startmult+ after one cycle of simulation

(defthm lemma_fact-cycle5
   (implies (equal (len st) 20)
            (equal (nth 15 (fact-cycle st))
```

```
                  (if (equal (nth 17 st) 1)
                      (if (equal (nth 18 st) 1)
                          (nth 15 st)
                          1)
                      (if (equal (nth 17 st) 2)
                          (if (equal (nth 9 st) 1)
                              0
                              (nth 15 st))
                          (nth 15 st)))))
  :hints (("Goal"
           :in-theory (disable fact.doit-cycle fact.multiplier-cycle))))

;;; The value of doit.r after one cycle of simulation

(defthm lemma_fact-cycle6
   (implies (equal (len st) 20)
            (equal (nth 18 (fact-cycle st))
                   (if (equal (nth 17 st) 2)
                       (if (equal (nth 9 st) 1)
                           (1- (nth 18 st))
                           (nth 18  st))
                       (if (equal (nth 17 st) 0)
                           (nth 0 st)
                           (nth 18 st))))))
  :hints (("Goal"
           :in-theory (disable fact.doit-cycle fact.multiplier-cycle))))

;;; The value of doit.f after one cycle of simulation

(defthm lemma_fact-cycle7
  (implies (equal (len st) 20)
   (equal (nth 19 (fact-cycle st))
  (if (equal (nth 17 st) 2)
      (if (equal (nth 9 st) 1)
  (nth 7 st)
(nth 19 st))
    (if (equal (nth 17 st) 0)
1
      (nth 19 st)))))
  :hints (("Goal"
           :in-theory (disable fact.doit-cycle fact.multiplier-cycle))))

;;; The value of doit.mystate after one cycle of simulation

(defthm  lemma_fact-cycle8
  (implies (equal (len st) 20)
   (equal (nth 17 (fact-cycle st))
```

```
  (if (equal (nth 17 st) 0)
      (if (equal (nth 1 st) 1) 1 0)
    (if (equal (nth 17 st) 1)
(if (equal (nth 18 st) 1) 0 2)
      (if (equal (nth 17 st) 2)
  (if (equal (nth 9 st) 1)
      1 2)
(nth 17 st))))))
  :hints (("Goal"
           :in-theory (disable fact.doit-cycle fact.multiplier-cycle))))

;;; The value of resmult+ after one cycle of simulation

(defthm intermed1
  (implies (and (equal (len st) 20)
(equal (nth 8 (fact.doit-cycle st)) 1)
(not (equal (nth 8 st) 1)))
   (equal (* (nth 5 st) (nth 6 st))
  (nth 14 st))))

(defthm intermed2
  (implies (and (equal (len st) 20)
(not (equal (nth 8 (fact.doit-cycle st)) 1))
(equal (nth 8 st) 1))
   (equal (nth 14 st)
  (* (nth 5 st) (nth 6 st)))))

(defthm lemma_fact-cycle9
  (implies (equal (len st) 20)
   (equal (nth 14 (fact-cycle st))
  (if (equal (nth 8 st) 1)
      (* (nth 5 st) (nth 6 st))
    (nth 14 st))))
  :hints (("Goal"
           :in-theory (disable fact.doit-cycle fact.multiplier-cycle))))

;;; The value of endmult+

(defthm lemma_fact-cycle10
  (implies (equal (len st) 20)
   (equal (nth 16 (fact-cycle st))
  (nth 8 (fact.doit-cycle st))))
  :hints (("Goal"
           :in-theory (disable fact.doit-cycle fact.multiplier-cycle))))

; ------------------------------------------------
;;; This function represents one step of computation ;;;
```

```
;;; A basic computation takes 3 simulation cycles    ;;;
; -------------------------------------------------

(defun computation-step (st)
    (fact-cycle (fact-cycle (fact-cycle st))))

; ----------------------------------------
;;; Basics properties
; ----------------------------------------

(defthm length_computation-step_lemma
  (equal (len (computation-step st))
 (len st))
  :hints (("Goal"
   :in-theory (disable fact-cycle))))

; ----------------------------------------
;   State properties about computation-step
; ----------------------------------------

;;; The value of doit.mystate after one cycle of computation

(defthm inter1
  (implies (and (equal (len st) 20)
(equal (nth 17 st) 1)
(equal (nth 8 st) 0)
(not (equal (nth 18 st) 1)))
   (equal (nth 8 (fact.doit-cycle (fact-cycle st)))
  1)))

(defthm lemma_computation1
  (implies (and (equal (len st) 20)
(equal (nth 17 st) 1)
(equal (nth 8 st) 0))
   (equal (nth 17 (computation-step st))
  (if (equal (nth 18 st) 1)
      (if (equal (nth 1 st) 1)
  (if (equal (nth 0 st) 1)
      0
    2)
0)
    1)))
  :hints (("Goal"
   :in-theory (disable fact-cycle fact.doit-cycle fact.multiplier-cycle))))

;;; The value of startmult after one cycle of computation
```

```
(defthm lemma_computation2
  (implies (and (equal (len st) 20)
(equal (nth 17 st) 1)
(equal (nth 8 st) 0)
(not (equal (nth 18 st) 1)))
   (equal (nth 8 (computation-step st))
  0))
  :hints (("Goal"
   :in-theory (disable fact-cycle  fact.doit-cycle fact.multiplier-cycle))))

;;; The value of doit.r after one cycle of simulation

(defthm lemma_computation4
  (implies (and (equal (len st) 20)
(equal (nth 17 st) 1)
(equal (nth 8 st) 0))
   (equal (nth 18 (computation-step st))
  (if (equal (nth 18 st) 1)
      (nth 0 st)
    (1- (nth 18 st)))))
  :hints (("Goal"
   :in-theory (disable fact-cycle))))

;;; The value of doit.f after one cycle of computation

(defthm lemma_computation5
  (implies (and (equal (len st) 20)
(equal (nth 17 st) 1)
(equal (nth 8 st) 0))
   (equal (nth 19 (computation-step st))
  (if (equal (nth 18 st) 1)
      1
    (* (nth 18 st) (nth 19 st)))))
  :hints (("Goal"
   :in-theory (disable fact-cycle))))

; ------------------------------------------------------------------------
;;; This function runs recursively and calls computation-step on doit.r
; ------------------------------------------------------------------------

(defun execute (st)
  (declare (xargs :measure (nfix (nth 18 st))
                  :hints (("Goal"
   :in-theory (disable computation-step)))))
  (if (and (integerp (nth 18 st))
   (equal (len st) 20)
   (equal (nth 17 st) 1)
```

```
   (equal (nth 8 st) 0))
      (if (< (nth 18 st) 2)
          (nth 19 st)
(execute (computation-step st)))
    st))

;;;  Example :
;;;  ACL2 !>(execute '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 7 1))
;;;  5040

(defun factorial (n)
  (if (zp n)
      1
    (* n (factorial (1- n)))))

; -----------------------------------------------------------------
;;;  Final correctness theorem
; -----------------------------------------------------------------

(defthm equivalence_fact_execute
  (implies (and (integerp (nth 18 st))
(equal (len st) 20)
(equal (nth 17 st) 1)
(equal (nth 8 st) 0)
(<= 2  (nth 18 st)))
   (equal (execute st)
  (* (nth 19 st)
     (factorial (nth 18 st)))))
  :hints (("Goal"
   :in-theory (disable computation-step))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; File "mult.lisp"
;;
;; generated by vhdl-Acl2 v.1.0 01/2001
;; P. Georgelin - TIMA Lab.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(in-package "ACL2")

(include-book "/home/preuves/georgelin/src/vhdl-acl2/utils")

(include-book "/home/preuves/georgelin/src/vhdl-acl2/expander")



;;; Function get-nth : gives the position of an element
;;; in the memory state

(defun MULT-GET-NTH (var)
   (declare (type (member req a b mult-state prod count done c done+ c+)
var)
    (xargs :guard t))
   (the (integer 0 9)

(case var ('req 0)
      ('a 1)
      ('b 2)
      ('mult-state 3)
      ('prod 4)
      ('count 5)
      ('done 6)
      ('c 7)
      ('done+ 8)
      ('c+ 9)))
)


;; Accessor of the memory state : (getst var st) -> value

(defun MULT-GETST (var st)
  (declare (xargs :guard t)
   (type (member req a b mult-state prod count done c done+ c+) var)
(type (satisfies true-listp) st))
  (nth (MULT-GET-NTH var) st))
```

```
;; Updater of the memory state : (putst ver new st) -> st

(defun MULT-PUTST (var new st)
  (declare (xargs :guard t)
   (type (member req a b mult-state prod count done c done+ c+) var)
(type (satisfies true-listp) st))
  (update-nth (MULT-GET-NTH var) new st))


;;;;;;;;;;;;;;


(defun MULT-STP (st)
 (declare (xargs :guard t))

(and (true-listp st)
     (= (length st) 10)
     (bitp (mult-getst 'req st))
     (signedp (mult-getst 'a st))
     (signedp (mult-getst 'b st))
     (signedp (mult-getst 'mult-state st))
     (signedp (mult-getst 'prod st))
     (signedp (mult-getst 'count st))
     (bitp (mult-getst 'done st))
     (signedp (mult-getst 'c st))
     (bitp (mult-getst 'done+ st))
     (signedp (mult-getst 'c+ st))))




;;;;;;;  This macro make-state creates an initial state ;;;;;

(defmacro MULT-MAKE-STATE
(&key (req '0)
      (a '0)
      (b '0)
      (mult-state '0)
      (prod '0)
      (count '0)
      (done '0)
      (c '0)
      (done+ '0)
      (c+ '0))
(list 'quote (list req a b mult-state prod count done c done+ c+)))
```

```
;;;;;  This function updates memory state ;;;;

(defun MULT-UPDATE-ST (key-arg st)
  (update-state-body "MULT" key-arg st))

;; This function represents the process MAIN ;;;;;;;

(defun MULT-MAIN-CYCLE (st)
(seq
  st
  (if (= (mult-getst 'mult-state st) 0)
      (seq st
           (if (= (mult-getst 'req st) 1)
               (seq st (mult-putst 'prod 0 st)
                    (mult-putst 'count
                                (mult-getst 'a st)
                                st)
                    (mult-putst 'mult-state 1 st))
               st))
      (seq st
           (if (= (mult-getst 'mult-state st) 1)
               (seq st
                    (if (= (mult-getst 'req st) 0)
                        (seq st (mult-putst 'mult-state 0 st))
                        (seq st
                             (if (> (mult-getst 'count st) 0)
                                 (seq st
                                      (mult-putst 'prod
                                                  (+ (mult-getst 'prod st)
                                                     (mult-getst 'b st))
                                                  st)
                                      (mult-putst 'count
                                                  (1- (mult-getst 'count st))
                                                  st))
                                 (seq st
                                      (mult-putst 'c+
                                                  (mult-getst 'prod st)
                                                  st)
                                      (mult-putst 'done+ 1 st)
                                      (mult-putst 'mult-state 2 st))))))
               (seq st
                    (if (= (mult-getst 'mult-state st) 2)
                        (seq st
                             (if (= (mult-getst 'req st) 0)
                                 (seq st (mult-putst 'c+ 0 st)
                                      (mult-putst 'done+ 0 st)
                                      (mult-putst 'mult-state 0 st))
```

```
                              st))
                        (seq st (mult-putst 'mult-state 0 st)))))))))


;;======== Function update of signals  =====


(defun MULT-UPDATE-SIGNALS (st)
(seq st
     (mult-putst 'done
                 (mult-getst 'done+ st)
                 st)
     (mult-putst 'c (mult-getst 'c+ st) st)))




;;======== Function cycle of simulation  =====


(defun MULT-CYCLE (st)
 (seq st (mult-main-cycle st)))



;;======== Function for N simulation cycles  =====


(defun MULT-SIMUL (n st)
   (if (zp n)
      st
    (mult-simul (1- n) (mult-cycle (mult-update-signals st)))))




:comp t
```

```
;; Proof of correctness of mult.lisp
;; P. Georgelin - July 2000


(in-package "ACL2")

(ld "mult.lisp")

;; the type of mult-putst is true-list when
;; st is a true-listp. This theorem is necessary
;; to satisfy hypothesis of all theorems using mult-putst
(defthm true-listp_mult-putst
    (implies (true-listp st)
             (true-listp (mult-putst var value st)))
:hints (("Goal" :in-theory (enable mult-putst mult-getst))))


; Simulation step maps well-formed states to well-formed states.
;(defthm mult-cycle-is-well-formed
;  (implies (mult-stp st)
;    (mult-stp (mult-cycle st))))


; Unfold one step of simulation function.
(defthm mult-unfold
  (and (equal (mult-simul 0 st) st)
       (implies (naturalp n)
(equal (mult-simul (1+ n) st)
       (mult-simul n (mult-cycle (mult-update-signals st))))))
  :hints (("Goal" :in-theory (disable mult-cycle))))


; Q.E.D 0.07 sec

;; and the results are put inside the following theorem
(defthm symb_simul_of_mult
    (implies (and (true-listp st)
                  (mult-stp st))
             (equal (mult-cycle st)
    (cond ((equal (mult-getst 'mult-state st) 0)
   (if (mult-getst 'req st)
       (mult-putst 'prod 0
       'count (mult-getst 'a st)
       'mult-state 1
       st)
     st))
  ((equal (mult-getst 'mult-state st) 1)
```

```
   (cond ((not (mult-getst 'req st))
  (mult-putst 'mult-state 0 st))

 ((> (mult-getst 'count st) 0)
  (mult-putst 'prod (+ (mult-getst 'prod st)
 (mult-getst 'b st))
  'count (1-
(mult-getst 'count st))
  st))
 (t (mult-putst 'c (mult-getst 'prod st)
    'done t
    'mult-state 2
    st))))
  ((equal (mult-getst 'mult-state st) 2)
   (if (not (mult-getst 'req st))
       (mult-putst 'c 0
       'done nil
       'mult-state 0
       st)
     st))
  (t (mult-putst 'mult-state 0 st))))))




; Q.E.D 0.13 sec


;; This function represents the computation cycle
;; the measure of this function is the value of signals mult-count
(defun computation-step (st)
   (declare (xargs :measure (acl2-count (mult-getst 'count st))
   :hints (("Goal"  :in-theory (disable
mult-cycle )))))
     (if (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st) (equal (mult-getst 'mult-state st) 1))

         (if (zp (mult-getst 'count st))
     st
      (computation-step (mult-cycle st)))
       st))


; Q.E.D 0.11 sec
```

```
#|
;Example

;ACL2 !>(computation-step
;                 (mult-make-state :req t :b 3 :st 1 :count 4))
; (T 0 3 NIL 0 1 12 0)

|#

;; This theorem proves the correctness of computation-step
;; It proves that value of signal mult-prod is the
;; multiplication between mult-count and mult-b
;; (+ the previous value of mult-prod if the first
;; value was not 1).
(defthm correctness_of_computation-step
    (implies (and (true-listp st)
  (mult-getst 'req st)
                 (mult-stp st)
  (equal (mult-getst 'mult-state st) 1)
  (>= (mult-getst 'count st) 0)
  (>= (mult-getst 'b st) 1))
            (equal (mult-getst 'prod (computation-step st))
    (+ (* (mult-getst 'count st)
  (mult-getst 'b st))
       (mult-getst 'prod st))))
   :hints (("Goal"  :in-theory (disable  mult-cycle))))

; Q.E.D 1.27 sec


(defthm mult-done_at_the_end_of_computation_step
    (implies (and (true-listp st) (mult-getst 'req st)
                 (mult-stp st) (equal (mult-getst 'mult-state st) 1)
  (>= (mult-getst 'count st) 0))
            (equal (mult-getst 'done
       (mult-cycle (computation-step st)))
   T))
   :hints (("Goal"  :in-theory (disable mult-cycle))))

; Q.E.D 1.11


(defthm mult-count_at_the_end_of_computation_step
    (implies (and (true-listp st) (mult-getst 'req st)
                 (mult-stp st) (equal (mult-getst 'mult-state st) 1)
  (>= (mult-getst 'count st) 0))
            (equal (mult-getst 'count (mult-cycle (computation-step st)))
```

```
   0))
   :hints (("Goal"  :in-theory (disable mult-cycle))))

; Q.E.D 1.14 sec


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; These two theorems unfold computation-step in a rewrite rule

(defthm computation-step-unfold
    (implies (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st) (equal (mult-getst 'mult-state st) 1)
    (> (mult-getst 'count st) 0))
       (equal (computation-step st)
              (computation-step (mult-cycle st))))
:hints (("Goal" :in-theory (disable mult-simul mult-cycle))))

; Q.E.D 0.29 sec


(defthm computation-step-unfold2
    (implies (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st) (equal (mult-getst 'mult-state st) 1)
    (= (mult-getst 'count st) 0))
       (equal (computation-step st)
             st))
:hints (("Goal" :in-theory (disable mult-simul mult-cycle))))

; Q.E.D 0.06 sec

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



(defthm mult-simul_unfold_left
  (implies (integerp n)
           (equal (mult-simul n (mult-cycle st))
                  (mult-cycle (mult-simul n st))))
:hints (("Goal" :in-theory (disable mult-cycle))))

; Q.E.D 0.07 sec


(defthm mult-unfold2
  (and (equal (mult-simul 0 st) st)
       (implies (and (naturalp n) (> n 0))
(equal (mult-simul  n st)
```

```
       (mult-simul (1- n) (mult-cycle st)))))
  :hints (("Goal" :in-theory (disable mult-cycle mult-unfold))))

; Q.E.D 0.04 sec


;; This theorem proves equivalence between computation-step and mult-simul

(defthm equiv_mult-simu_computation-step
   (implies (and (true-listp st)
 (mult-getst 'req st)
                (mult-stp st)
 (equal (mult-getst 'mult-state st) 1))
     (equal (computation-step st)
    (mult-simul (mult-getst 'count st) st)))
:hints (("Goal" :in-theory (disable mult-cycle))))

; Q.E.D 1.40 sec


(defthm equiv_mult-simu_computation-step+step
   (implies (and (true-listp st)
 (mult-getst 'req st)
                (mult-stp st)
 (equal (mult-getst 'mult-state st) 1))
     (equal (mult-cycle (computation-step st))
    (mult-cycle (mult-simul (mult-getst 'count st) st))))
:hints (("Goal" :in-theory (disable mult-cycle))))

; Q.E.D 0.07 sec


;; -----------------------------------------------------------------
;; Proof of correctness of the component for 'count
;; simulation cycle
;; -----------------------------------------------------------------


;; after 'count simulation cycle, 'prod contains
;; the product

(defthm theorem_mult-simul_is_multiplication
    (implies (and (true-listp st)
 (mult-getst 'req st)
                (mult-stp st)
 (equal (mult-getst 'mult-state st) 1)
(>= (mult-getst 'count st) 1) (>= (mult-getst 'b st) 1))
```

```
      (equal (mult-getst 'prod (mult-simul
 (mult-getst 'count st) st))
    (+ (* (mult-getst 'count st) (mult-getst 'b st))
       (mult-getst 'prod st))))

:hints (("Goal" :in-theory (disable mult-simul mult-cycle)
         :use (equiv_mult-simu_computation-step
      correctness_of_computation-step))))


; Q.E.D 0.17 sec


;; done is T after the computation step + 1
(defthm theorem_done_is_T
   (implies (and (true-listp st)
 (mult-getst 'req st)
                 (mult-stp st)
 (equal (mult-getst 'mult-state st) 1))
     (equal (mult-getst 'done (mult-simul
(1+ (mult-getst 'count st)) st))
    T))
:hints (("Goal" :in-theory (disable mult-cycle)
         :use (mult-done_at_the_end_of_computation_step))))

; Q.E.D 0.35 sec


;; If the request is false when mult-state is 0
;; then there is no computation
(defthm not_request
    (implies (and (true-listp st)
  (mult-stp st)
  (= (mult-getst 'mult-state st) 0) ;initial state
  (not (mult-getst 'req st)))
     (equal (mult-cycle st)
st)))

; Q.E.D 0.04 sec

;; When done becomes t, result is mult-prod
(defthm result_unchanged_when_done
  (implies (and (true-listp st)
(mult-stp st)
(equal (mult-getst 'mult-state st) 1)
(equal (mult-getst 'count st) 0)
```

```
(not (mult-getst 'done st))
(mult-getst 'done (mult-cycle st)))
   (equal (mult-getst 'prod (mult-cycle st))
  (mult-getst 'prod st))))

; Q.E.D 0.07 sec


;; When not request occurs, the next value of mult-state
;; will be 0 (initalisation state)
(defthm if_not_request_occurs
  (implies (and (true-listp st)
(mult-stp st)
(not (mult-getst 'req st)))
   (equal (mult-getst 'mult-state (mult-cycle st))
  0)))


; Q.E.D 0.10 sec


;; If done becomes t then the next value of done
; is conditionned by the value of mult-req
(defthm when_done_is_nil
  (implies (and (true-listp st)
(mult-stp st)
(not (mult-getst 'done st))
(mult-getst 'done (mult-cycle st)))
   (equal (mult-getst 'done (mult-simul 2 st))
  (if (not (mult-getst 'req (mult-simul 2 st)))
      nil
      (mult-getst 'done (mult-cycle st))))))

; Q.E.D 0.87 sec


;; A rewrite rule for initialisation of internals signals
(defthm lemma_state_0
   (implies (and (true-listp st)
 (mult-stp st)
 (mult-getst 'req st)
 (equal (mult-getst 'mult-state st) 0))
    (and (equal (mult-getst 'prod (mult-cycle st)) 0)
 (equal (mult-getst 'count (mult-cycle st))
(mult-getst 'a st))
 (equal (mult-getst 'mult-state (mult-cycle st))
1)
```

```
 (mult-stp (mult-cycle st))
 (mult-getst 'req (mult-cycle st))
 (true-listp (mult-cycle st)))))


(defthm lemma_mult-simul
  (implies (and (true-listp st)
 (mult-stp st)
 (mult-getst 'req st)
 (equal (mult-getst 'mult-state st) 0))
   (equal (mult-simul (+ 2 (mult-getst 'a st)) st)
  (mult-simul (1+ (mult-getst 'a (mult-cycle st)))
      (mult-cycle st))))
:hints (("Goal"
 :in-theory (disable mult-cycle))))


; Q.E.D 0.29 sec


;; Final theorem for done :
;; done becomes T after a + 2 simulation cycle

(defthm total_result_done
  (implies (and (true-listp st)
 (mult-stp st)
 (mult-getst 'req st)
 (equal (mult-getst 'mult-state st) 0))
   (equal (mult-getst 'done (mult-simul
(+ 1 (mult-getst 'count (mult-cycle st)))
(mult-cycle st)))
  T))
:hints (("Goal"  :do-not-induct t
         :in-theory (disable mult-cycle
     mult-simul
     mult-unfold2
     mult-unfold
     SYMB_SIMUL_OF_MULT
     MULT-SIMUL_UNFOLD_LEFT)
                :use (:instance theorem_done_is_T (st (mult-cycle st))))))

(defthm total_result_done2
  (implies (and (true-listp st)
 (mult-stp st)
 (mult-getst 'req st)
 (equal (mult-getst 'mult-state st) 0))
   (equal (mult-getst 'done (mult-simul
```

```
(+ 1 (mult-getst 'a st))
(mult-cycle st)))
  T))
:hints (("Goal"  :do-not-induct t
         :in-theory (disable mult-cycle
     mult-simul
     mult-unfold2
     mult-unfold
     SYMB_SIMUL_OF_MULT
     MULT-SIMUL_UNFOLD_LEFT)
                :use (total_result_done))))


; Q.E.D 0.31 sec


;; starting from state 0 we obtain the multiplication after
;; a + 1 simulation cycle

(defthm inputs-unchanged
  (implies (and (true-listp st)
(mult-stp st))
   (equal (mult-getst 'b (mult-cycle st)) (mult-getst 'b st))))

(defthm total_result_prod
  (implies (and (true-listp st)
 (mult-stp st)
 (mult-getst 'req st)
 (equal (mult-getst 'mult-state st) 0)
 (>= (mult-getst 'a st) 1) (>= (mult-getst 'b st) 1))
   (equal (mult-getst 'prod (mult-simul
(mult-getst 'count (mult-cycle st))
(mult-cycle st)))
  (* (mult-getst 'count (mult-cycle st)) (mult-getst 'b st))))
:hints (("Goal"  :do-not-induct t
         :in-theory (disable mult-cycle
     mult-simul
     mult-unfold2
     mult-unfold
     SYMB_SIMUL_OF_MULT
     MULT-SIMUL_UNFOLD_LEFT)
                :use (:instance theorem_mult-simul_is_multiplication (st (mult-cycle st))))))

(defthm total_result_prod2
  (implies (and (true-listp st)
 (mult-stp st)
 (mult-getst 'req st)
```

```
 (equal (mult-getst 'mult-state st) 0)
 (>= (mult-getst 'a st) 1) (>= (mult-getst 'b st) 1))
   (equal (mult-getst 'prod (mult-simul
(mult-getst 'a st)
(mult-cycle st)))
  (* (mult-getst 'a st) (mult-getst 'b st))))
:hints (("Goal"  :do-not-induct t
         :in-theory (disable mult-cycle
     mult-simul
     mult-unfold2
     mult-unfold
     SYMB_SIMUL_OF_MULT
     MULT-SIMUL_UNFOLD_LEFT)
                :use total_result_prod)))


;; When done becomes t then the value of mult-c takes the value of mult-prod
(defthm mult-prod_and_mult-c
  (implies (and (true-listp st)
(mult-stp st)
(mult-getst 'req st)
(not (mult-getst 'done st))
(equal (mult-getst 'mult-state st) 1))
   (equal (mult-getst 'c (mult-cycle st))
  (if (mult-getst 'done (mult-cycle st))
      (mult-getst 'prod st)
      (mult-getst 'c st))))
:hints (("Goal" :in-theory (disable mult-simul mult-cycle))))

; Q.E.D 0.24 sec


;;; lemmas about the value of state at the end

(defthm mult-state_at_the_end_of_computation_step
    (implies (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st)
  (equal (mult-getst 'mult-state st) 0)
  (>= (mult-getst 'count st) 0))
             (equal (mult-getst 'mult-state
       (mult-cycle (computation-step st)))
    1))
   :hints (("Goal"  :in-theory (disable mult-cycle))))

(defthm mult-state_at_the_end_of_computation_step2
    (implies (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st)
```

```
  (equal (mult-getst 'mult-state st) 1)
  (>= (mult-getst 'count st) 0))
             (equal (mult-getst 'mult-state
      (computation-step st))
    1))
   :hints (("Goal"  :in-theory (disable mult-cycle)
    :use (mult-state_at_the_end_of_computation_step))))


(defthm mult-state_at_the_end_of_mult-simu
    (implies (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st)
  (equal (mult-getst 'mult-state st) 1)
  (>= (mult-getst 'count st) 0))
             (equal (mult-getst 'mult-state
       (mult-simul (mult-getst 'count st) st))
    1))
   :hints (("Goal"  :in-theory (disable mult-cycle)
    :use (mult-state_at_the_end_of_computation_step2))))

(defthm mult-state_at_the_end_of_mult-simu2
    (implies (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st)
  (equal (mult-getst 'mult-state st) 0)
  (>= (mult-getst 'count st) 0))
             (equal (mult-getst 'mult-state
        (mult-simul (mult-getst 'count (mult-cycle st)) (mult-cycle st)))
    1))
   :hints (("Goal"  :in-theory (disable mult-cycle)
    :use (:instance mult-state_at_the_end_of_mult-simu (st (mult-cycle st))))))

;;; lemmas about the value of done at the end

;(defthm mult-done_at_the_end_of_computation_step
;    (implies (and (true-listp st) (mult-getst 'req st)
;                   (mult-stp st) (equal (mult-getst 'mult-state st) 1)
;   (>= (mult-getst 'count st) 0))
;              (equal (mult-getst 'done
;        (mult-cycle (computation-step st)))
;    T))
;   :hints (("Goal"  :in-theory (disable mult-cycle))))

(defthm mult-done_at_the_end_of_computation_step2
    (implies (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st)
  (equal (mult-getst 'mult-state st) 1)
  (equal (mult-getst 'done st) nil)
```

```
  (>= (mult-getst 'count st) 0))
             (equal (mult-getst 'done
       (computation-step st))
    nil))
   :hints (("Goal"  :in-theory (disable mult-cycle EQUIV_MULT-SIMU_COMPUTATION-STEP))))

(defthm mult-done_at_the_end_of_mult-simul
    (implies (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st)
  (equal (mult-getst 'mult-state st) 1)
  (equal (mult-getst 'done st) nil)
  (>= (mult-getst 'count st) 0))
             (equal (mult-getst 'done
      (mult-simul (mult-getst 'count st) st))
    nil))
   :hints (("Goal"  :in-theory (disable mult-cycle)
    :use (mult-done_at_the_end_of_computation_step2 EQUIV_MULT-SIMU_COMPUTATION-STEP))))

(defthm mult-done_at_the_end_of_mult-simul2
    (implies (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st)
  (equal (mult-getst 'mult-state st) 0)
  (equal (mult-getst 'done st) nil)
  (>= (mult-getst 'count st) 0))
             (equal (mult-getst 'done
        (mult-simul (mult-getst 'count (mult-cycle st)) (mult-cycle st)))
    nil))
   :hints (("Goal"  :in-theory (disable mult-cycle)
    :use (:instance mult-done_at_the_end_of_mult-simul (st (mult-cycle st))))))



(defthm total_result_done2
  (implies (and (true-listp st)
 (mult-stp st)
 (mult-getst 'req st)
 (equal (mult-getst 'mult-state st) 0))
   (equal (mult-getst 'done (mult-simul
(+ 1 (mult-getst 'a st))
(mult-cycle st)))
  T))
:hints (("Goal"  :do-not-induct t
         :in-theory (disable mult-cycle
     mult-simul
     mult-unfold2
     mult-unfold
     SYMB_SIMUL_OF_MULT
```

```
     MULT-SIMUL_UNFOLD_LEFT)
                 :use (total_result_done))))

(defthm mult-done_at_the_end_of_mult-simul3
    (implies (and (true-listp st) (mult-getst 'req st)
                  (mult-stp st) (equal (mult-getst 'mult-state st) 1)
  (>= (mult-getst 'count st) 0))
             (equal (mult-getst 'done
       (mult-cycle (mult-simul (mult-getst 'count (mult-cycle st))
      (mult-cycle st))))
   T))
   :hints (("Goal"  :in-theory (disable mult-cycle)
    :use (mult-done_at_the_end_of_computation_step EQUIV_MULT-SIMU_COMPUTATION-STEP))))


(defthm total_result_c
  (implies (and (true-listp st)
 (mult-stp st)
 (mult-getst 'req st)
 (equal (mult-getst 'mult-state st) 0)
 (equal (mult-getst 'done st) nil)
 (>= (mult-getst 'a st) 1)
 (>= (mult-getst 'b st) 1))
   (equal (mult-getst 'c (mult-cycle (mult-simul
      (mult-getst 'a st)
      (mult-cycle st))))
  (* (mult-getst 'a st) (mult-getst 'b st))))
:hints (("Goal"  :do-not-induct t
         :in-theory (disable mult-cycle
     mult-simul
     mult-unfold2
     mult-unfold
     SYMB_SIMUL_OF_MULT
     MULT-SIMUL_UNFOLD_LEFT)
                 :use ((:instance total_result_prod2 (st (mult-simul (mult-getst 'a st)
     (mult-cycle st))))
       mult-state_at_the_end_of_mult-simu2
       mult-done_at_the_end_of_computation_step
       mult-done_at_the_end_of_mult-simul2
       mult-prod_and_mult-c))))
```

# Bibliographie

[1] Mark Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-Johan H. Seger. A methodology for large-scale hardware verification. In *Formal Method In Computer-Aides Design*, pages 263–282, 2000.

[2] Mark Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Design Automation Conference*, pages 538–541, 1998. citeseer.nj.nec.com/aagaard98combining.html.

[3] Inc Advanced Micro Device. Amd athlon processor model 1 and model 2 revision guide. http://www.amd.com/, August 2000.

[4] R. Airiau, J.-M. Bergé, V. Olive, and J. Rouillard. *VHDL du langage à la modélisation.* Presses polytechniques et universitaires romandes et CNET-ENST, 1990.

[5] Laurent Arditi. *BMD Can Delay the Use of Theorem Proving for Verifying Arithmetic Assembly Instructions. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 34–48, Pala Alto, CA, USA, November 1996. Springer.

[6] Bruno Barras, Samuel Boutin, Cristina Cornes, Judica Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Fillietre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Henri Laulhere, Cesar Mueoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saebi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual: Version 6.1.* IEEE Synthesis Interoperability W.G. 1076.6, December, 1997.

[7] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In Jr W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided design*, volume 1954, pages 3–36. Springer, 2000.

[8] Valeria Bertacco, Maurizio Damiani, and Stefano Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In *Design Automation Conference*, pages 391–396, 1999. citeseer.nj.nec.com/bertacco99cyclebased.html.

[9] D. Borrione and P. Georgelin. Formal verification of VHDL using VHDL-like ACL2 models. In *Proceedings of Forum on Design Languages (FDL'99)*, pages 105–116, Ecole Normale Superieure de Lyon, Lyon, France, 1999.

[10] D. Borrione, P. Georgelin, and V. Rodrigues. Symbolic simulation and verification of VHDL with ACL2. In *International Conference on HDL (HDLCONF'2000)*, pages 167–182, San Jose, 2000.

[11] D. Borrione, P. Georgelin, and V. Rodrigues. Using macros to mimic VHDL. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies*, pages 167–183. Kluwer Academic Press, 2000.

[12] Dominique Borrione, Julia Dushina, and Laurence Pierre. Formalization of finite state machines with data path for the verification of high-level synthesis. In *XI Brazilian Symposium on Integrated Circuit Design (SBCCI'98)*, Buzios, Rio de Janeiro, Brazil, 1998. IEEE Computer Society.

[13] Dominique Borrione and Ashraf Salem. Denotational semantics of a synchronous vhdl subset. In *Formal Methods in System Design*, volume 7, Number 1/2, pages 53–71, August 1995.

[14] Dominique Borrione, editor. Special issue on VHDL semantics. *Formal Methods in System Design*, 7(1/2), 1995.

[15] R. S. Boyer, D. Goldschlag, M. Kaufmann, and J S. Moore. Functional instanciation in first order logic, 1991. In Artifical Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy. Editors: V. Lifschitz. Academic Press, 1991.

[16] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, 1979.

[17] Robert S. Boyer and J Strother Moore. *Automated Reasoning and Its Applications*, chapter Mechanized formal reasonning about programs and computing machines. MIT Press, 1996.

[18] Peter T. Breuer, Luis Sanchez Fernandez, and Carlos Delgado Kloos. A Simple Denotational Semantics, Proof Theory and a Validation Condition Generator for Unit-Delay VHDL. In *Formal Methods in System Design*, volume 7, Number 1/2, pages 27–51, August 1995.

[19] B. Brock, M. Kaufmann, and J S. Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, 1996.

[20] R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, pages 688–694, Los Alamitos, Ca., USA, 1985. IEEE Computer Society Press.

[21] Randal E. Bryant and Yirng-An Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. In *Proceedings of 32nd Design Automation Conference (DAC'95)*, pages 535–541, San Francisco, CA, USA, 1995.

[22] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[23] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *Proceedings of International Conference on Computer-Aided Verification*, volume LCNS 818. Springer Verlag, June 1994.

[24] A. J. Camilleri. A role for theorem proving in multi-processor design. In *Proceedings of CAV'98*, pages 275–293, Vancouver, June 1998. Springer-Verlag LNCS N 1427.

[25] P. Camurati and P. Prinetto. Formal verification of hardware correctness: Introduction and survey of current reseach. In *Computer (Journal)*, volume 21(7), pages 9–19, July 1988.

[26] Ritu Chadha. Symbolic simulation: Theory and application to protocol modeling and validation. citeseer.nj.nec.com/65745.html.

[27] Edmund Clarke, Somesh Jha, and Dong Wang. Abstract bdds: A technique for using abstraction in model checcking. In Laurence Pierre and Thomas Kropf, editors, *Proceedings of Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703. Springer Verlag, 1999.

[28] E.M. Clarke, M. Khaira, and X. Zhao. Word Level Model Checking- Avoiding the Pentium FDIV Error. In *Proceedings of 33rd Design Automation Conference (DAC'96)*, pages 645–648, Las Vegas, NV, USA, 1996.

[29] Intel Corp. Fdiv replacement program information. http://support.intel.com/.

[30] Intel Corp. Pentium iii processor specification update. ftp://download.intel.com/.

[31] Michel Cosnard and Denis Trystram. *Algorithmes et architectures paralleles*. Informatique intelligence artificielle. InterEditions, Paris, 1993.

[32] John A. Darringer and James C. King. Applications of symbolic execution to program testing. *Computer*, 11(4):51–60, 1978.

[33] Nancy A. Day, Jeffrey R. Lewis, and Byron Cook. Symbolic simulation of microprocessor models using type classes in Haskell. Technical Report CSE-99-005, Department of Computer Science, Oregon Graduate Institute, June 1999.

[34] D. Deharbe. *Verification formelle de proprietes temporelles: etude et application au langage VHDL*. PhD thesis, Universite Joseph Fourier, Grenoble, France, 1996.

[35] Carlos Delgado Kloos and Peter T. Breuer, editors. *Formal Semantics for VHDL*. Kluwer, 1995.

[36] Julia Dushina. *Vérification Formelle des Résultats de la Synthèse de Haut Niveau*. PhD thesis, Université Joseph Fourier, Grenoble, 1999.

[37] D. D. Gajski, J. Zhu, R. Domer, A. Gerslauer, , and S. Zhao. *VHDL du langage à la modélisation*. Kluwer Academic Publishers, 2000.

[38] P. Georgelin. Vhdl-acl2, see http://philippe.georgelin.online.fr/vhdl-acl2, 2001.

[39] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL; A Theorem Proving Environment for Higher Order Logic*. Cambridge University, Cambridge, 1993.

[40] David Greve and Matthew Wilding. Two handy update-nth equality rules. Draft version - short note - Rockwell Collins Inc.

[41] D. Hardin, M. Wilding, and D. Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle, In Alan J. Hu and Moshe Y. Vardi, editors, Computer-Aided Verification, CAV '98, volume 1427 of Lecture Notes in Computer Science, pages 39–44, Vancouver, Canada, June 1998. Springer-Verlag.

[42] M. Harrand, J. Sanchez, A. Bellon, J. Bulone, A. Tournier, O. Deygas, J.C Herluison, D. Doise, and E. Berrebi. A single-chip cif 30-hz, h261, h263, and h263+ video encoder/decoder with embedded display controller. In *IEEE Journal of solid-state circuits*, Nov. 1999.

[43] John Harrison. Formal verification of floating point trigonometric functions. In Jr W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954, pages 217–233. Springer, 2000.

[44] IEEE Standards Board. *IEEE Std 1076-1993 VHDL Language Reference Manual*. The Institute of Electrical and Electronics Engineers, Inc, New York, USA, September 15 1993.

[45] IEEE Synthesis Interoperability W.G. 1076.6. *IEEE Standard VHDL Subset for Behavioral Syntheses (working document)*, 2000. http://www.eda.org/siwg.

[46] Motorola Inc. Mpc7400 part number specification. http://e-www.motorola.com/.

[47] J.A. Darringer. The application of program verification techniques to hardware verification. In *Proceedings of the Sixteenth ACM/IEEE Design Automation Conference*, pages 375–381, Los Alamitos, 1979. IEEE Computer Society Press.

[48] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[49] M. Kaufmann, P. Manolios, and J S. Moore. *Computer Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[50] M. Kaufmann and J S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Transactions on Software Engineering*, 23:203–213, April 1997.

[51] M. Kaufmann and D. Russinoff. Verification of pipeline circuits. In Matt Kaufmann and J Strother Moore, editors, *ACL2 Workshop 2000 Procedings*. University of Texas at Austin, 2000.

[52] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology. citeseer.nj.nec.com/article/kaufmann96acl.html.

[53] K. Keutzer. The need for formal methods for integrated circuit design. In *Proceedings of FMCAD'96*, pages 1–18, Palo Alto, CA, Nov. 1996. Springer-Verlag LNCS N 1166.

[54] T. Lynch and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the amd5k86 floating-point division program. In *IEEE Transactions on Computers*, volume 47(9), September 1998.

[55] Panagiotis Manolios. Mu-calculus model-checking. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[56] Pete Manolios. Verification of pipelined machines in acl2. In Matt Kaufmann and J Strother Moore, editors, *ACL2 Workshop 2000 Procedings*. University of Texas at Austin, 2000.

[57] John W. O'Leary Mark D. Aagaard, Thomas F. Melham. Xs are for trajectory evaluation, booleans are for theorem proving (extended version). citeseer.nj.nec.com/268533.html.

[58] Inc Matt Kaufmann Computational Logic. The expander book. ftp://ftp.cs.utexas.edu/pub/moore/acl2/v2-5/acl2-sources/books/cli-misc/expander.lisp, 1997.

[59] K. C. McMillan. *Symbolic Model Checking*. Kluwer, Boston, 1993.

[60] K.L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Proceedings of Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703. Springer Verlag, 1999.

[61] J S. Moore. Symbolic simulation: An ACL2 approach. In *FMCAD'98*, pages 334–350, 1998. LNCS 1522.

[62] J Strother Moore. Rewriting for symbolic execution. Technical report, Department of Computer Sciences, University of Texas at Austin, 2000.

[63] Olaf Müller and Tobias Nipkow. *Combining Model Checking and Deduction for I/O-Automata*. 1995. http://www4.informatik.tu-muenchen.de/papers/MuellerNipkow_TaAf1995.html.

[64] S. Narain. Reasoning about hybrid systems with symbolic simulation, 1994. Narain, S.: Reasoning about hybrid systems with symbolic simulation. Invited paper, Proceedings of 11th International Conference on Analysis and Optimization of Systems. Guy Cohen and Jean-Pierre Quadrat (eds.), Lecture Notes in Control and Information Sciences 199 (1994).

[65] S. Narain and R. Chadha. Symbolic discrete-event simulation, 1994. S. Narain, R. Chadha, Symbolic Discrete-Event Simulation, Discrete-Event Systems, Manufacturing Systems and Communication Networks, Editors: P.R. Kumar and P. Varaiya, LNCS, Springer Verlag 1994.

[66] F. Nicoli. *Verification formelle de descriptions VHDL comportementales*. PhD thesis, Universite de Provence, Marseille, France, 1999.

[67] 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, June 1992. Springer-Verlag.

[68] Laurence Pierre. Induction-oriented verification of replicated architectures described in vhdl. *Journal of Circuits, Systems and Computers*, 10(3- 4):147–189, 2000.

[69] Simon Read and Martyn Edwards. A Formal Semantics of VHDL in Boyer-Moore Logic. In *Conference on Concurrent Engineering and EDA (CEEDA)*, Poole, Great Britain, 1994.

[70] J. Reed, J. Sinclair, and F. Guigand. Deductive reasoning versus model checking: two formal approaches for system development, 1999. J.N. Reed, J.E. Sinclair, and F. Guigand. Deductive reasoning versus model checking: two formal approaches for system development. In K. Taguchi K. Araki, A. Galloway, editor, Integrated Formal Methods 1999, York, UK, June 1999. Springer Verlag.

[71] Gerd Ritter. Sequential equivalence checking by symbolic simulation. In *Formal Methods in Computer-Aided Design FMCAD2000*, volume LNCS 1954. Springer Verlag, November 2000.

[72] Gerd Ritter. *Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation*. PhD thesis, Université Joseph Fourier and Darmstadt University of Technology, 2001.

[73] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998.

[74] D. Russinoff. A case study in formal verification of register-transfer logic with acl2: The floating point adder of the amd athlon processor. In Jr W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided design*, volume 1954, pages 3–36. Springer, 2000.

[75] David M. Russinoff. A Formalization of a Subset of VHDL in the Boyer-Moore Logic. In *Formal Methods in System Design*, volume 7, pages 7–25, August 1995.

[76] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 84–97, Liege, Belgium, 1995. Springer Verlag. citeseer.nj.nec.com/332791.html.

[77] J. Sanchez. *Specification of IVT Reconstruction operator*, Feb, 2000. internal report, STMicroelectronics, Sarl.

[78] Jun Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, December 1999. Also available from http://www.cs.utexas.edu/users/sawada/-dissertation/diss.html.

[79] Jun Sawada and Jr Warren A. Hunt. Hardware modeling using function encapsulation. In Jr Warren A. Hunt and steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*. Springer, 2000.

[80] K. Schneider and M. Huhn. Comparing model-checking and term-rewriting in the verification of an embedded system, 1999.

[81] N. Shankar. PVS: Combining specification, proof checking and model checking. In *FMCAD'96*, pages 257–264, 1996. LNCS 1166.

[82] G. L. Steele, Jr. *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA 01803, second edition, 1990. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/clm/clm.html.

[83] Deborah G. Tatar. *A Programmer's Guide to Common Lisp*. Digital Press, 1987.

[84] P. Georgelin V. Rodrigues, D. Borrione. An acl2 model of vhdl for symbolic simulation and formal verification. *XIII Symposium on Integrated Circuits and Systems Design (SBCCI'00),Manaus, Amazonas, Brazil*, September 18-22, 2000.

[85] Jr. Warren A. Hunt. The de language. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[86] M. M. Wilding, D. A. Greve, and D. S. Hardin. Efficient simulation of formal processor models. Technical report, Advanced Technology Center, Rockwell Collins Avionics and Communications, Cedar Rapids, IA 52498, 1998. http://pobox.com/users/hokie/docs/efm.ps.

[87] Matthew M. Wilding. Robust computer system proofs in pvs. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM97: Fourth NASA Langley Formal Methods Workshop*. NASA, NASA Conference Publication no. 3356, 1997.

[88] C. Wilson, D. L. Dill, and R. E. Bryant. Symbolic simulation with approximate values. In Jr. W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer Aided Design FMCAD '2000*, volume LNCS 1954, pages pp. 486–504, November 2000.

## Résumé

La plupart des outils de vérification formelle comme les "Model-checkers" sont restrictifs car ils ne peuvent travailler avec des niveaux plus haut que le "RTL", et ils sont également limités sur le nombre total d'états. Les démonstrateurs de théorèmes ne souffrent pas de ces restrictions, mais ne sont pas automatiques et requièrent des méthodes pour faciliter leur utilisation systématique. Cette thèse aborde la vérification formelle de descriptions VHDL au moyen du démonstrateur ACL2. Nous proposons un environnement combinant simulation symbolique et démonstrateur de théorèmes pour l'analyse formelle de descriptions de haut niveau d'abstraction.

Plus précisément, notre approche consiste à développer des méthodes
- pour formaliser un sous-ensemble de VHDL,
- pour "diriger" le démonstrateur pour effectuer de la simulation symbolique
- pour utiliser ces résultats pour les preuves.

Un outil a été développé combinant des traducteurs (VHDL vers ACL2), des moteurs de simulation symbolique et de preuves, et une interface utilisateur. Les définitions et les théorèmes sont générés automatiquement. Un même modèle généré est ainsi utilisé pour toutes les tâches. Nous aspirons à fournir au concepteur une méthodologie pour insérer la vérification formelle le plus tôt possible dans le cycle de conception. Le démonstrateur est utilisé pour des manipulations symboliques et pour prouver qu'ils sont équivalents à une fonction spécifiée.

Le résultat de cette thèse est de rendre la technique de démonstration de théorèmes acceptable dans une équipe de concepteur du point de vue de la facilité d'utilisation, et de diminuer le temps de vérification.


**Mot Clés:** Vérification formelle, démonstration de théorèmes, Acl2, simulation symbolique.



## Abstract

To satisfy market requirements, formal verification tools must allow designers to verify complex descriptions and reason about large or infinite sets of values. One should be able to concentrate on the correctness of algorithms and the essential mathematical properties of the blocks being designed.

Most modern verification tools such as Model Checkers are restrictive because they can't deal with abstraction levels higher than Register Transfer Level, or similar Finite-State Machine models and are also limited on the total number of states. Theorem provers do not suffer from these restrictions, but they are not fully automated, and require methods to ease their systematic use in the standard design flow. This thesis addresses the formal verification of VHDL descriptions with the ACL2 theorem prover. We propose an environment combining symbolic simulation and theorem proving for the formal analysis of high level VHDL designs. Our approach consists in developping methods
- to formalize a synthesis subset of VHDL,
- to "direct" the theorem prover to perform symbolic simulation
- to use symbolic simulation results for proofs.

A tool was developped combining translators from VHDL to ACL2, symbolic simulation and proof engines in a user interface. The definitions and theorems that formalize the VHDL input are generated automatically, and the resulting model is executable. This same model is used for symbolic simulation and proof. By combining symbolic simulation and theorem proving, we aim at providing the verification engineer with a methodology to efficiently insert formal verification in the very early specification stages of a design. The theorem prover can be used to perform symbolic manipulations on the result expressions, and prove that they are equivalent to a specified function. The result of this thesis is to make theorem proving techniques more acceptable to a design team in terms of ease of use, and to notably decrease verification time in a design process.


**Keyword:** formal verification, theorem prover, Acl2, symbolic simulation.

ISBN 2 - 913 329 - 73 - X Broché
ISBN 2 - 913 329 - 74 - 8 Version électronique