Rapport final du marché 99-34-024
Vérification de programmes VHDL

E. Encrenaz-Tiphène E. Paviot-Adet
D. Poitrenaud
Laboratoire d'Informatique de Paris VI

Chapitre 1

Introduction

Ce document constitue le rapport final (fourniture 6) du marché 99-34-024 - *Extension des diagrammes de décision binaire pour le traitement des types finis. Application à la vérification de systèmes complexes.*

Le modèle retenu est le langage VHDL pour la description des systèmes et la logique temporelle arborescente pour la spécification des propriétés. Le type d'algorithme de vérification visé est la vérification symbolique de modèle (*symbolic model checking*).

Le point central des algorithmes de ce type est la représentation concise d'ensembles d'états et la réalisation d'opérations efficaces sur ces ensembles. ces opérations doivent permettre le calcul de l'ensemble des états successeurs (et prédécesseurs) d'un ensemble d'états. La structure de donnée généralement retenue est les diagrammes binaires de décision.

Des travaux antérieurs au projet actuel ont montré que cette structure de donnée est un bon support pour un sous-ensemble du langage VHDL. Toutefois, une limitation importante de ces travaux est que les programmes pris en compte ne devaient faire intervenir que des variables booléennes et ne pas contenir de contraintes temporelles (clause `for` des instructions de suspension par exemple).

Le problème principal auquel notre étude devait répondre est la prise en compte de variables ayant un type fini et en particulier les contraintes temporelles. Le codage par des diagrammes de décision binaire pour ce type de variable n'est pas satisfaisant. En effet, dans le cas favorable où les bornes des domaines sont connues *a priori*, les opérations de l'arithmétique entière redéfinies en terme d'opérations booléennes nuisent aux performances et dans le cas le moins favorable pour lequel les bornes sont inconnues, ceci implique l'introduction dynamique de nouvelles variables booléennes en cours de traitement.

L'objectif du projet était de construire un outil de vérification sur une nouvelle structure partagée permettant une prise en compte directe de variables de domaine fini. Cette nouvelle structure - les diagrammes de décision de données (DDD) - a été définie par le LaBRI. Une des caractéristiques importantes de cette structure est la possibilité de définir des opérations de manipulation étant

propres au domaine d'application.

Le travail de l'équipe du Laboratoire Informatique de Paris 6 a été la définition d'un outil de vérification de programmes VHDL basé sur les DDD. Ce travail a consisté en un choix de représentation des états du programme sous la forme d'un DDD et la définition de l'ensemble des opérations nécessaires au calcul de l'ensemble des états successeurs (et prédécesseurs) d'un ensemble d'états. Un premier prototype a été réalisé. Il couvre une part importante du langage et permet la vérification de propriétés de sûreté. Ces fonctionnalités sont en effet suffisantes pour démontrer la validité de notre approche.

Le prototype est constitué d'une bibliothèque, nommée `VHDL-DDD`, permettant le calcul de l'ensemble des états stables d'un programme VHDL.

Cette bibliothèque est composée d'un ensemble d'outils permettant :

- de représenter des programmes VHDL manipulant des variables et signaux de type entier, ainsi que des instructions de suspension *temporelle* à l'aide des Diagrammes de Décision et de Données développés par nos partenaires du LaBRI ;
- de calculer l'ensemble des états stables du programme précédemment représenté ;
- de sélectionner les sous-ensemble d'états stables satisfaisant une propriété.

De plus, nous avons réalisé un programme mettant en œuvre la bibliothèque `VHDL-DDD`. Ce programme produit le code C++ d'un vérificateur pour un code VHDL fourni en entrée. Le code VHDL peut contenir des instructions particulières permettant la spécification des propriétés devant être vérifiées.

La suite de ce document est composée comme suit :

- Le chapitre 2 est consacrée, au travers d'un exemple didactique, à la découverte de l'usage de la bibliothèque `VHDL-DDD` et plus particulièrement de la manière dont un programme VHDL est décrit au sein de cette bibliothèque. Une dernière section présente brièvement l'outil de vérification construit au dessus de cette bibliothèque.
- Le chapitre 3 précise l'implémentation de la bibliothèque `VHDL-DDD` et les opérations de manipulation de DDD qui y sont mis en place.
- Enfin, le dernier chapitre dresse un bilan du projet, présente une première évaluation de l'outil de vérification produit et propose des perspectives de travail.

Des informations complémentaires (un manuel de référence de la bibliothèque, une description de son architecture interne, la grammaire retenue pour l'outil de vérification et un ensemble d'exemples) sont fournies en annexe.

Chapitre 2

Utilisation de la bibliothèque VHDL-DDD

Dans ce chapitre, nous montrons comment la bibliothèque `VHDL-DDD` doit être utilisé pour représenter un programme VHDL. Cette présentation est réalisée au travers d'un exemple simple. La représentation du programme peut ensuite être employée pour calculer l'ensemble des états stables du programme et vérifier des propriétés d'accessibilité. Ces deux dernières fonctionnalités ainsi qu'un programme mettant en oeuvre cette bibliothèque sont présentés à la fin de ce chapitre. Un manuel de référence présentant toutes les classes C++ employées dans ce chapitre est donné dans l'annexe A.

Des hypothèses concernant la syntaxe du programme VHDL ont été définies.

En premier lieu la bibliothèque `VHDL-DDD` considère que le programme est *élaboré* (i.e. toute instruction concurrente est transformée en son processus équivalent) et donc qu'il est uniquement composé d'une collection de processus concurrents naviguant dans un océan de signaux.

Les appels de procédures ou fonctions sont considérés comme étant expansés (i.e. chaque appel est remplacé par le code de la fonction ou la procédure correspondante) et donc ne sont pas explicitement représentés.

Généralement, deux modèles de mise à jour de la valeur effective des signaux d'entrée sont possibles. Dans le premier de ces modèles, les signaux d'entrée peuvent changer à tous les cycles de simulation (c'est le cas le plus général). Dans le second, le programme est réactif, et dans ce cas le programme atteindra toujours un état stable avant de prendre en considération de nouvelles valeurs sur les signaux d'entrée. Nous proposons une modélisation des signaux d'entrée pour ces deux modèles de comportement. Il est important de noter que la construction de l'ensemble des états accessibles stables présuppose que le programme est réactif et donc que le premier modèle ne pourra être employé que dans un programme de vérification portant sur les états transitoires du système, ce qui n'est pas le cas ici.

Ces premières hypothèses ne contraignent pas la classe des programmes VHDL que nous pouvons écrire. Les hypothèses suivantes limitent quant à elles la classe des programmes que nous analysons.

Les signaux disposent d'un seul pilote (i.e. ils ne peuvent être affectés que par un processus). Ceci implique que nous ne traitons pas de fonction de résolution. De plus, l'affectation est immédiate (c'est à dire prise en compte au delta cycle suivant) : la clause `AFTER` de l'affectation d'un signal n'est pas considérée.

Les seuls types de données (des signaux et des variables) que nous prenons en compte sont des entiers bornés. Les types énumérés (en particulier les booléens) et les types enregistrement (même imbriqués) peuvent être simulés par une ou plusieurs variables entières. Par contre, les variables réelles et les tableaux ne seront pas couverts ici.

Dans l'état actuel de la bibliothèque, ces trois dernières limitations ne sont pas levées, car elles impliquent une modélisation complexe du pilote d'un signal (le pilote devient une liste ordonnée de couples indiquant la valeur à affecter et le délai résiduel potentiel, et les affectations de tels signaux consistent en des manipulations parfois complexes de ces listes prenant en compte des fonctions de résolution). Toutefois, il est important de noter que les DDD ne posent aucun problème technique à leur prise en compte.

2.1 Exemple de programme VHDL

La présentation de `VHDL-DDD` est illustrée sur le petit programme suivant.

```
architecture A of E is

constant MAX: integer := 15;
signal S1, S2 : integer range 0..MAX := 0;

begin

P1 : process
        variable V1 : integer := 1;
        begin
        S1 <= V1;
        wait on S1,S2 until S1/=0 or S2/=0 and V1/=0 for 10 ns;
        S1 <= (S1 + 1) mod MAX;
        V1 := S2;
        wait for 15 ns;
end process P1;
```

```
P2 : process
        begin
        S2 <= (MAX+(S2 − 1) mod MAX) mod MAX;
        wait on S1 for 5 ns;
        wait on S2 for 10 ns ;
        S2 <= (S2 + 2) mod MAX;
        wait on S1;
end process P2;

end A;
```

Cet exemple ne représente pas un composant existant et est purement académique. Il a été construit pour mettre en lumière le style de programmes VHDL que nous considérons (il satisfait les restrictions précédentes) et pour illustrer différents cas d'affectations de variables et signaux de type entier, ainsi que différentes configurations de suspension. En effet,

- le processus $P1$ dispose d'une variable locale $V1$ qui interagit avec les deux signaux $S1$ et $S2$ (respectivement en lecture et en écriture).
- le programme effectue des affectations de variable (l'affectation prend effet dès l'exécution de l'instruction) et de signaux (l'affectation prend effet lors du prochain cycle de simulation).
- les principales formes de suspension sont illustrées :
  - suspension sur une liste de sensibilité seule (3e `wait` du processus $P2$).
  - suspension sur un délai uniquement (2e `wait` du processus $P1$ et 2e `wait` du processus $P2$ - dans ce dernier cas, le signal présent dans la liste de sensibilité ne sera jamais activé lors de l'exécution de l'instruction de suspension).
  - suspension sur liste de sensibilité et délai résiduel (1e instruction `wait` du processus $P2$).
  - suspension sur liste de sensibilité, clause `until` et délai résiduel (1e instruction `wait` du processus $P1$).

2.2 Codage des signaux

Les signaux sont implémentés dans la bibliothèque par trois classes C++ `SignalIn`, `Signal` et `SignalOut`. Il est à noter que dans la version actuelle de la bibliothèque, un signal est soit un signal purement d'entrée (et donc associé à aucun pilote), soit un signal interne (et potentiellement de sortie) associé à un unique pilote, soit un signal purement de sortie. Les trois classes citées ci-dessus permettent respectivement la déclaration de signaux de ces trois catégories. De plus, un signal est déclaré comme appartenant à un programme. Nous commençons donc par introduire la classe C++ `Program` et le constructeur qui lui est associé.

Le constructeur de la classe `Program` prend un unique paramètre, le nom du

programme. Voici un exemple de déclaration d'un objet de type `Program`.

```
Program prog("My Prog");
```

Une fois un programme déclaré, il est possible de définir les signaux qui le composent. Les trois constructeurs de signaux prennent en paramètre une référence vers le programme dont le signal dépend, ainsi que son nom et les bornes de son domaine de définition. De plus, un signal interne est paramétré par sa valeur initial.

Pour notre programme d'exemple, la déclaration des deux signaux $S1$ et $S2$ est la suivante :

```
Signal  s1(prog, "S1", 0, 0, 15),
        s2(prog, "S1", 0, 0, 15);
```

2.3 Codage des processus

Un processus est composé de variables locales et d'un ensemble d'instructions. Ces instructions manipulent des expressions faisant intervenir les signaux (ainsi que leurs attributs `EVENT` et `TRANSACTION`), les variables locales et des constantes. Les processus, les variables locales, les expressions et les attributs sont implémentés sous la forme de classes C++ (respectivement `Process`, `Variables`, `Expression` et `Attribut`) alors que les instructions sont définies par le biais de fonctions.

Parmi les instructions composant le corps d'un processus, les instructions de suspension sont traitées de façon particulière. En effet, l'exécution de ce type d'instruction est réalisée dans des phases de simulation distinctes des autres instructions. De plus, nous distinguons le traitement d'évaluation de la condition de réveil des instructions de suspension, de celui de leur exécution proprement dite.

Lorsqu'un processus atteint une instruction de suspension comportant une clause `for`, l'alarme de ce processus doit être armée avec le délai spécifié dans la clause. Pour des raisons de facilité et d'homogénéisation, la bibliothèque impose que ce traitement soit matérialisé explicitement par une instruction particulière d'affectation de l'alarme qui précède chaque instruction de suspension disposant d'une clause `for`.

Avant d'introduire le codage retenu pour les instructions d'un processus, revenons un instant sur notre programme d'exemple. Après avoir inséré les instructions matérialisant la mise à jour des alarmes des processus (elles apparaissent en commentaire), nous devons associer à chaque instruction la valeur du compteur ordinal lui correspondant.

P1 : **process**
begin
L11: S1 <= V1;
L12: -- armer l'alarme de P1 à 10

L13: **wait on** S1,S2 **until** S1/=0 **or** S2/=0 **and** V1/=0 **for** 10 ns;
L14: S1 <= (S1 + 1) **mod** MAX;
L15: V1 := S2;
L16: −− armer l'alarme de P1 à 15
L17: **wait for** 15 ns;
end process P1;

P2 : **process**
begin
L21: S2 <= (MAX+(S2 − 1) **mod** MAX) **mod** MAX;
L22: −− armer l'alarme de P2 à 5
L23: **wait on** S1 **for** 5 ns;
L24: −− armer l'alarme de P2 à 10
L25: **wait on** S2 **for** 10 ns ;
L26: S2 <= (S2 + 2) **mod** MAX;
L27: **wait on** S1;
end process P2;

Nous pouvons remarquer que la dernière instruction de suspension du processus $P2$ ne comporte pas de clause `for` et qu'en conséquence l'alarme du processus n'est pas armée. La manipulation des valeurs des compteurs ordinaux dans un programme est facilitée par la définition d'un type énuméré tel que :

```
enum {L11=11,L12,L13,L14,L15,L16,L17,
      L21=21,L22,L23,L24,L25,L26,L27};
```

2.3.1 Processus et variables locales

Le constructeur de la classe `Process` est paramétré par une référence vers le programme dans lequel est déclaré le processus, le nom du processus ainsi que la valeur initiale de son compteur ordinal. Les processus $P1$ et $P2$ de notre programme d'exemple sont déclarés comme suit :

```
Process p1(prog, "P1", L11),
        p2(prog, "P2", L21);
```

La déclaration d'une variable locale est très similaire à celle d'un signal. En effet, une variable est caractérisée par une référence vers le processus dans lequel elle est déclarée, son nom, sa valeur initiale et les bornes de son domaine de définition. La déclaration de la variable $V1$ locale au processus $P1$ est réalisé de la manière suivante :

```
Variable v1(p1, "V1", 1, MININT, MAXINT);
```

où `MININT` et `MAXINT` représentent les bornes minimale et maximale du type prédéfini `INTEGER`. Il est à noter que la valeur initiale d'une variable ne peut être qu'une valeur entière constante (et non pas une expression constante).

2.3.2 Attributs et expressions

Les deux seuls attributs VHDL supportés actuellement par la bibliothèque sont les attributs `EVENT` et `TRANSACTION` associés aux signaux. La seule façon d'obtenir un objet de la classe C++ `Attribut` est d'employer l'une des deux méthodes `event()` et `transaction()` de la classe `Signal`. Nous verrons par la suite que les objets du type `Attribut` peuvent être employés pour la construction des listes de sensibilité des instructions de suspension ainsi que pour la définition d'expressions.

Les instructions composant un processus ont pour point commun de manipuler des expressions (pour désigner la valeur devant être affectée ou la condition d'une instruction conditionnelle). Une première version d'un gestionnaire d'expression a été mise en oeuvre dans la bibliothèque.

Dans la version actuelle, une expression est implémentée par un objet de type `Expression`. Le seul constructeur (autre que le constructeur par copie) proposé par cette classe est paramétré par un entier et représente une expression constante. L'usage de la conversion implicite mise en place par le C++, nous permet de fournir directement des valeurs constantes (par exemple `1`, `MAX` ou encore `true` ou `false`) là où une expression est attendue.

Les autres moyens d'obtention d'un objet de type `Expression` sont des opérations de conversion (d'une variable, d'un signal ou d'un attribut vers une expression) ainsi que des opérations arithmétiques ou logiques. Ici encore, les conversions implicites sont possibles. Par exemple, dans notre programme, l'expression `(S1 + 1) mod MAX` intervenant dans l'instruction $L3$ peut être obtenue en écrivant `(s1+1)%MAX` (rappelons que `%` est l'opérateur C++ rendant le reste de la division entière). Cette expression est une expression entière. La bibliothèque ne gère que des opérations de type entier et la prise en compte d'expressions booléennes est simulée en considérant que toute expression ayant une valeur différente de 0 est vraie et qu'elle est fausse dans le cas contraire.

2.3.3 Instructions

Dans la version actuelle de la bibliothèque, cinq types d'instructions sont supportés :

- affectation d'une variable par une expression,
- affectation d'un signal par une expression,
- affectation d'une alarme par une expression,
- branchement conditionnel :
  - `if ... then ... else ... end if`
  - `while ... loop ... end loop`
  - `loop ... end loop`
  - `for... loop ... end loop`
- instruction de suspension.

A chaque type d'instruction correspond une fonction de la bibliothèque. La liste suivante présente le prototype de ces fonctions.

```
void assignVar(  Process& p, int cpc, int npc,
                 const Variable& v, const Expression& e);
void assignSig(  Process& p, int cpc, int npc,
                 const Signal& s, const Expression& e);
void assignAlarm(Process& p, int cpc, int npc,
                 const Expression& e);
void branch(     Process& p, int cpc, int npc_then, int npc_else,
                 const Expression& e);
void wait(       Process& p, int cpc, int npc, const SensList& s,
                 const Expression& e, bool b);
```

Le paramètre `p` référence le processus dans lequel l'instruction apparaît. Les paramètres `cpc` et `npc` désignent respectivement la valeur du compteur ordinal correspondant à l'instruction et la valeur atteinte par exécution de cette instruction. Dans le cas particulier d'une instruction de branchement conditionnelle, deux valeurs possibles peuvent être atteintes (`npc_then` et `npc_else`) en fonction du résultat de l'évaluation de l'expression. Il est à noter que cette dernière forme est suffisante pour coder l'ensemble des instructions de branchement conditionnel.

Enfin, les instructions d'affectation sont paramétrées par l'objet devant être affecté (implicite dans le cas d'une affectation d'alarme) et par l'expression proprement dite.

Les instructions de suspension font apparaître des listes de sensibilité. Les listes de sensibilité sont gérées par la classe C++ `SensList`. Les éléments composant une telle liste sont des attributs relatifs aux signaux ou des signaux proprement dits (ce dernier cas est équivalent à l'attribut `EVENT` du signal considéré). La fonction `insert` permet la constitution d'une liste de sensibilité :

```
SensList sl;
insert(sl, s1.event());
insert(sl, s2); // équivalent à insert(sl, s2.event());
insert(sl, s3.transaction());
```

L'expression intervenant dans les paramètres de la fonction `wait` est la condition de la clause `until`. Enfin, le dernier paramètre de cette fonction indique si l'alarme a été armée au préalable et donc si une condition temporelle doit être prise en compte pour le réveil du processus.

Le code complet correspondant à l'exemple de la section 2.1 est donné en annexe (section C.1.2). Il comprend la définition de toutes les expressions manipulées et des instructions de chacun des processus.

2.4 Vérification de programme

Une fois les signaux et les processus du programme `prog` définis, nous sommes à même de définir le simulateur et de provoquer le calcul de l'ensemble des états stables.

```
Simulator sim(prog);
DDD reach = sim.reachable();
```

L'ensemble `reach` des états stables est la base pour la définition d'un outil de vérification.

Dans la version actuelle de la bibliothèque, la seule fonction permettant de vérifier un programme est une fonction de sélection des états stable satisfaisant une propriété d'accessibilité. Elle peut être employée aussi bien pour vérifier que certains états sont bien atteints ou que des états interdits ne le sont pas. Le prototype de cette fonction est le suivant :

```
DDD select(const DDD& d, const Expression& e);
```

La propriété d'accessibilité est représentée par l'expression e. Cette expression est considérée comme étant booléenne et la fonction `select` renvoie tous les états représentés dans `d` rendant l'expression `e` satisfaite.

L'expression peut faire intervenir des constantes, des variables, des signaux et des attributs. De plus, pour permettre de prendre en compte la valeur de l'alarme et du compteur ordinal des processus, la classe `Process` propose deux méthodes ayant les prototypes suivants :

```
Expression al() const;
Expression pc() const;
```

Toutefois, il est à noter que la fonction `select` est employé sur un ensemble d'états stable et donc que le compteur ordinal d'un processus ne peut donc désigner qu'une instruction de suspension.

Pour l'exemple de la section 2.1, l'extrait de code suivant sélectionne l'ensemble des états stables du programme tel que la variable `V1` du premier processus a la valeur 1 alors que le deuxième processus est en attente sur la deuxième instruction de suspension.

```
DDD reach = sim.reachable();
DDD error = select(reach, (v1==1) && (p2.pc()==L25));
if (error.size() != 0)
   cout << "des erreurs ont été rencontrées" << endl;
else
   cout << "tous les états sont corrects" << endl
```

2.5 Le programme VerifVHDL

Une première version d'un outil de vérification a été développé. Il consiste en un compilateur acceptant un sous-ensemble du langage VHDL augmenté de la clause `assert` permettant la définition des états stables recherchés. Ce compilateur produit un code C++ basé sur la bibliothèque `VHDL-DDD` qui, une fois compilé, calcule l'ensemble des états stables du programme, sélectionne ceux satisfaisant au moins une des clauses `assert` apparaissant dans le programme et les affiche.

Cette section présente la syntaxe de la part du langage VHDL couverte par le prototype ainsi que la syntaxe retenus pour les clauses `assert`.

2.5.1 Vérification d'une propriété d'état

Un prototype de compilateur a été écrit de façon à pouvoir traduire rapidement un programme VHDL en un programme C++. La syntaxe se rapproche de celle des programmes élaborés : une entité dans laquelle les signaux en entrée et en sortie sont déclarés, suivie d'une architecture contenant les déclarations des signaux internes et les processus. Le but de cet outil étant d'obtenir une aide au développement d'exemples, certaines simplifications de la syntaxe de VHDL ont été opérées.

Les seuls types supportés sont les suivants :

- types énumérés,
- `integer`,
- sous-types (intervalles) du type `integer`,
- bit.

Les enregistrements ne sont pas supportés, les variables de ce type doivent donc être traduites en autant de variables que de champs dans l'enregistrement.

Les types d'opérations pouvant être effectuées par les processus sont les suivants :

- affectation de variables et signaux, la valeur affectée pouvant être le résultat d'un calcul,
- instruction de suspension comportant des clauses `on`, `until` et `for`.

Les structures de contrôle implémentées sont :

- `if-then`,
- `if-then-else`,
- `while-loop`.

Les conditions pouvant être des tests sur les valeurs de variables ou de signaux, ou bien encore des tests sur les attributs événement et transaction associés aux signaux. Seule restriction : la dernière instruction de la section `then` dans le cas d'une structure `if-then-else` doit être une instruction simple (instruction d'affectation ou de suspension).

2.5.2 Vérification d'une propriété d'état

Une propriété d'état est définie par une expression portant sur les variables d'état du programme VHDL. On suppose que tous les objets manipulés par le simulateur sont accessibles (notamment les variables locales d'un processus, son alarme, son compteur ordinal). Les noms des objets locaux à un processus sont préfixés par le nom du processus dans lequel ils sont déclarés.

La déclaration d'une propriété suit la syntaxe :

```
<etiquette_propriété> : assert(expression_booléenne);
```

`expression_booléenne` est telle que reconnue dans le parser. Les noms des objets apparaissant dans ces expressions suivent la grammaire suivante :

```
nom_objet ::=
        <nom_processus>.<nom_variable>
    |   <nom_processus>.pc           -- pc pour ``program counter''
    |   <nom_processus>.al
    |   <nom_processus>.<nom_signal> -- si interne à un processus
    |   <nom_signal>
```

Pour les compteurs ordinaux, la seule opération de comparaison autorisée est l'égalité à une étiquette d'instruction `wait`. Pour les alarmes des processus, la valeur de l'alarme peut être comparée à une expression évaluable en une valeur entière.

2.5.3 Vérification d'une propriété portant sur une séquence d'états

On peut étendre la vérification à des séquences d'états en introduisant des processus observateurs. Ces derniers sont des processus concurrents au même titre que tous les processus VHDL, mais leur comportement ne peut influer sur le système qu'ils observent. Les signaux observés les font progresser d'état en état, gardant ainsi la mémoire de la séquence exécutée. La vérification de propriété d'états peut s'appliquer aux états de ces processus observateurs, conduisant ainsi à la vérification de propriétés de séquences du système. Les observateurs étant des processus au sens de VHDL, les règles de visibilité du langage leurs sont appliquées : ils ne peuvent observer que des signaux globaux, et n'ont pas accès aux données privées des processus qu'ils observent.

La syntaxe de tels processus suit la syntaxe VHDL.

Un exemple d'expression de propriété d'état et de propriété de séquence est donné en fin du programme de la section C.2. Les résultats de l'évaluation de l'outil `VerifVHDL` sont résumé dans le chapitre 4.

Chapitre 3

Element de conception de la bibliothèque `VHDL-DDD`

La structure générale de la bibliothèque `VHDL-DDD` est schématisée dans la figure 3.1. Ainsi un objet de type `Program` a connaissance de l'ensemble des processus (objets de type `Process`) et des signaux (objets de type `Signal`) le composant. De même, un objet de type `Process` a accès aux différentes variables (objets de type `Variable`) et aux instructions (représentées par des homomorphismes forts - cf. rapport du LaBRI) le composant.

Avant de détailler le calcul des états stables d'un programme, nous précisons la manière dont est représenté un état de ce programme.

3.1 Codage d'un état d'un programme VHDL

L'état d'un programme est caractérisé par l'état local de chacun de ses signaux et de ses processus.

Sachant que les signaux sont mono-pilotés, l'état d'un signal est défini par :

Sa valeur effective. Cette variable intervient dans les expressions de condition ou à droite d'un opérateur d'affectation. Elle prend des valeurs dans le domaine des entiers (éventuellement un intervalle).

La valeur de son attribut événement. Cette variable correspond à l'attribut `event` du langage et elle prend des valeurs dans l'ensemble $\{0, 1\}$.

La valeur de son attribut transaction. Cette variable correspond à l'attribut `transaction` du langage et elle prend des valeurs dans l'ensemble $\{0, 1\}$.

Le calcul de la nouvelle valeur de l'attribut transaction après l'exécution d'un delta cycle, nous amène à introduire une variable auxiliaire permettant de déterminer si le signal a été affecté lors de la phase d'exécution du cycle de simu-

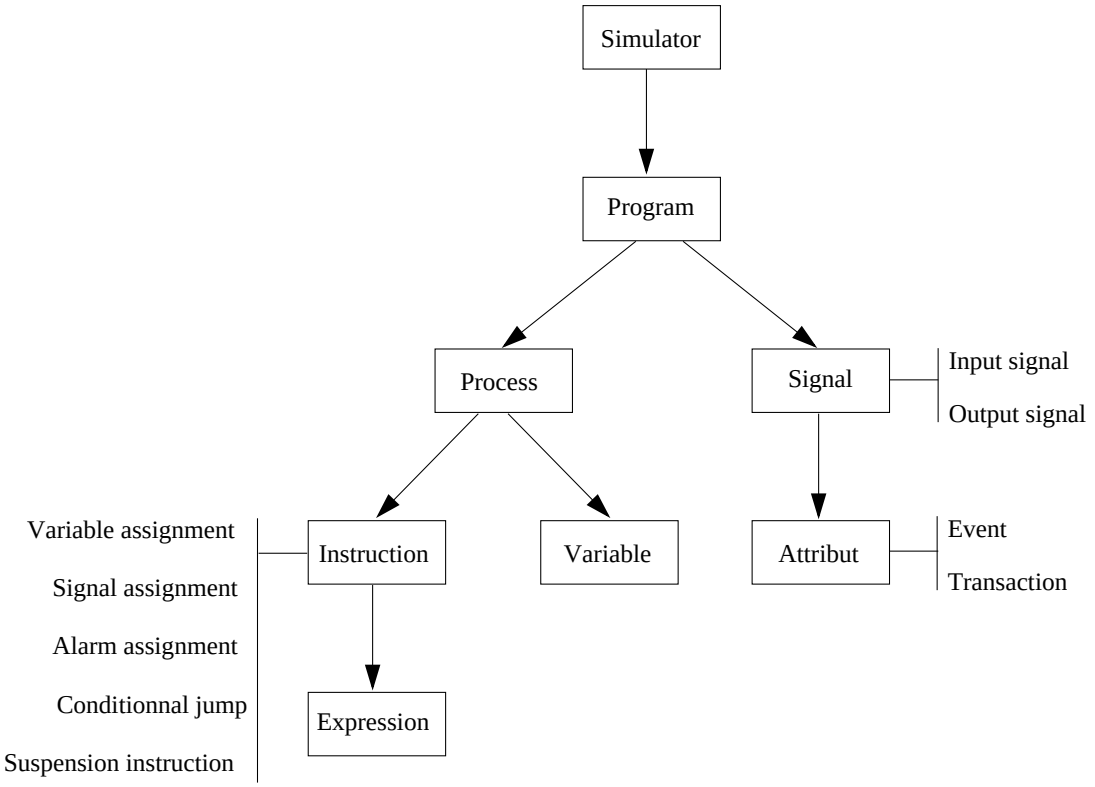

FIG. 3.1 – Structure du programme

lation courant. Cette variable prend elle aussi des valeurs dans l'ensemble $\{0, 1\}$.

L'état d'un processus est défini par :

La valeur de son compteur ordinal. Cette variable est un entier borné (type énuméré).

La valeur de son alarme. Cette variable correspond au délai résiduel de franchissement du processus lors d'attente sur des instructions `wait` ayant une clause temporelle `for`.

La valeur de chaque variable locale Chacune de ces variables est supposée avoir un domaine entier (éventuellement borné)

La valeur pilotée de chaque signal affecté. Cette variable est de même domaine que la variable représentant la valeur effective du signal. Elle correspond au nom du signal dans la partie gauche des instructions du processus réalisant une affectation de signal.

Enfin, nous définissons une dernière variable auxiliaire, nommée `WAKEUP`, étant globale au programme et permettant de distinguer si l'état considéré est *stable* (`WAKEUP`=0) ou non (`WAKEUP`=1). Il est à noter que cette variable n'est pas essentielle et est introduite uniquement dans un but d'optimisation du calcul des états stables.

Un ensemble d'états est représenté dans le simulateur par un objet de type `DDD`. A chaque variable sus-nommée est associée une variable `DDD` correspondante. La bibliothèque DDD offre des mécanismes pour la déclaration de variables ainsi que l'association à celles-ci d'un identificateur affichable. La bibliothèque `VHDL-DDD` se repose sur ces mécanismes pour déclarer toutes les variables nécessaires lors de la construction des objets correspondant (i.e. `Program`, `Signal`, `Process` et `Variable`). Les identificateurs affichables sont déterminés en fonction des paramètres des constructeurs correspondants.

Les valeurs initiales de chacune des variables impliquées dans l'état d'un programme est connue. Un signal (valeur effective et pilotée) ou une variable locale ne comportant pas d'initialisation explicite est initialisé à la valeur la plus à gauche du type. Les attributs des signaux sont tous initialisés à faux par défaut. Les valeurs initiales des compteurs ordinaux sont spécifiées par un des paramètres du constructeur de la classe `Process`. Enfin, la variable `WAKEUP` est initialisée à 0 (i.e. l'état initial du programme n'est pas un état stable).

La construction de l'état initial est réalisée par le biais de l'opérateur de concaténation des DDD. Il est à noter que l'ordre dans lequel sont concaténées les variables a une forte influence sur les performances de la bibliothèque.

Dans le prototype, nous avons imposé que la variable `WAKUP` soit la première puis soit suivie des caractéristiques des signaux puis des processus.

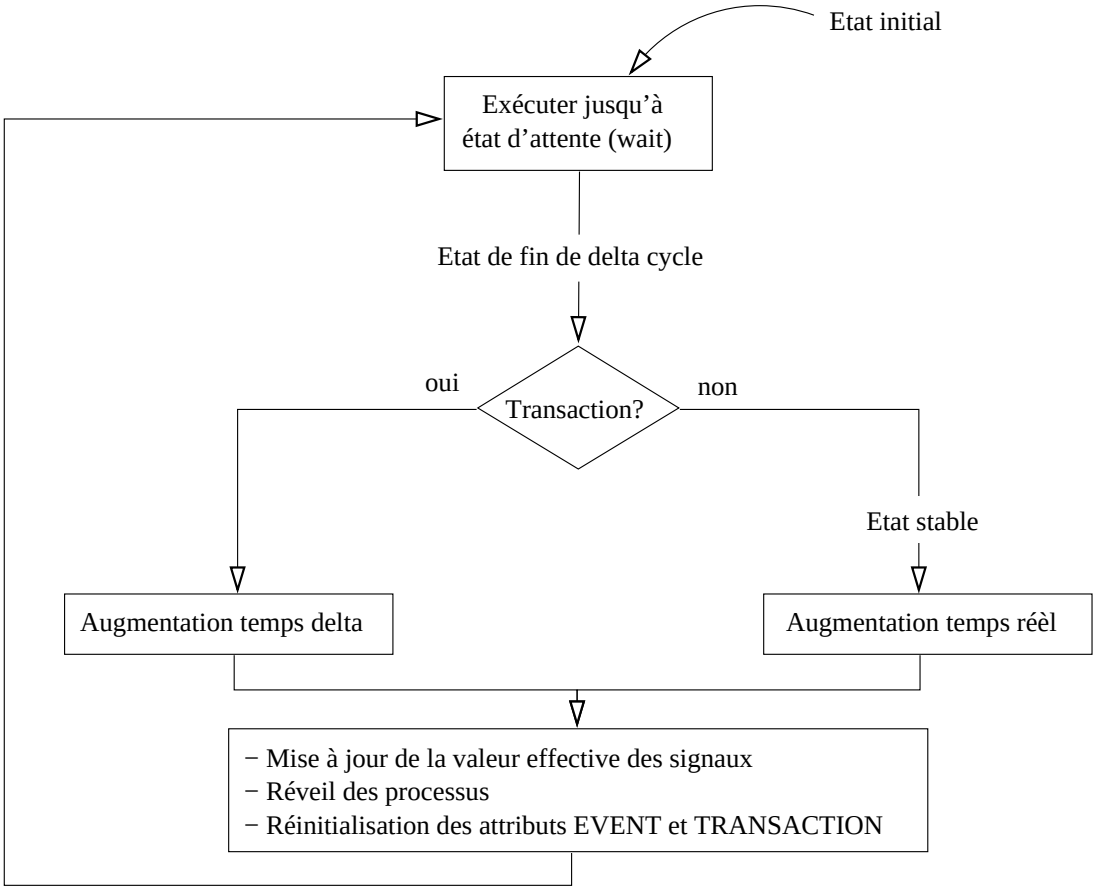

FIG. 3.2 – Automate de simulation

3.2 Cycle de simulation d'un programme VHDL

Cette section décrit le cycle complet de simulation mis en place lors du calcul de l'ensemble des états stables et explicite les différentes opérations réalisées à chaque instant du calcul.

Le calcul de l'ensemble des états accessibles reprend l'algorithme de simulation d'un programme VHDL et l'applique *simultanément* à un ensemble d'états du programme. Le cycle de simulation est rappelé sur le schéma de la figure 3.2.

Initialement, tous les processus sont prêts à exécuter leur première instruction. Ceci est représenté par la flèche nommée `état initial` menant dans la phase d'exécution représentée par le bloc `Exécuter jusqu'à état d'attente`.

Durant la phase d'exécution, les processus exécutent leurs instructions jusqu'à ce qu'ils soient suspendus (par l'exécution d'une instruction de suspension, `wait...`). Dans cette étape, chaque processus s'exécute indépendamment de la progression des autres processus : il n'y a pas de variable partagée, et les signaux, qui eux peuvent être lus et modifiés par plusieurs processus, distinguent la valeur

effective du signal (accessible en lecture uniquement par tous les processus) de celles de ses pilotes (un pilote de signal est accessible en écriture uniquement et pour un seul processus). Les valeurs écrites dans les pilotes seront reportées dans les valeurs effectives dans une autre étape du cycle de simulation.

Une fois tous les processus suspendus, les modifications provoquées par la phase d'exécution doivent être traitées pour pouvoir réamorcer l'exécution des processus. Ces modifications peuvent être à effet *immédiat* (doivent être prise en compte dans le même instant physique) ou *différé* (elles auront lieu à une date physique ultérieure). Cette alternative est représentée sur la figure 3.2 par le choix sur les variables `transaction`, qui conditionne le type d'augmentation du temps à réaliser par la suite. Deux cas se présentent.

- Si il y a des mises à jour de valeurs de signaux à effectuer au temps courant (elles résultent de l'exécution d'instructions d'affectations de signaux dans la phase d'exécution précédente, lesquelles ont positionné l'attribut `transaction` des signaux affectés à 1), le temps augmente d'un délai microscopique *delta*. Sur la figure 3.2, cela correspond à l'étape `Augmentation Temps Delta`.
- Si il n'y a pas de mise à jour à effectuer au temps courant (tous les attributs `transaction` sont à 0), l'état est considéré comme *stable*, et c'est le temps *réel* qui est augmenté. Ceci est représenté par l'étape `Augmenter Temps Réel` sur la figure 3.2. Dans cette étape, les délais de franchissement des processus qui ont été positionnés dans une phase d'exécution antérieure sont décrémentés jusqu'à ce que l'un d'entre eux (au moins) s'annule.

Une fois le temps augmenté, il y a mise à jour des valeurs effectives des signaux à partir des valeurs contenues dans leurs pilotes. Puis les processus qui étaient en attente sur une clause qui a été levée, soit du fait de l'augmentation du temps, soit du fait de la mise à jour d'un signal, sont réveillés. Ceci correspond au bloc `Mise à jour des valeurs effectives des signaux` et `Réveil des processus`. Les attributs des signaux sont réinitialisés (notamment l'attribut `transaction` qui sera positionné dans la phase d'exécution suivante). Alors les processus qui peuvent s'exécuter s'exécutent concurremment jusqu'à être de nouveau suspendus, dans une nouvelle étape d'exécution décrite dans le bloc `Exécuter jusqu'à état d'attente`

3.3 Homomorphismes forts d'un programme

Chacune des opérations citées ci-dessus est appliquée sur un ensemble d'états représenté sous la forme d'un DDD.

L'implémentation de ces opérations a été réalisée par le biais des homomorphismes forts de la bibliothèque développée au LaBRI. Les homomorphismes forts que nous avons défini (environ une vingtaine) ne font généralement aucune

hypothèse sur l'ordre des variables composant les DDD sur lesquels ils seront appliqués.

Toutefois cette règle générale souffre d'une exception. En effet, deux homomorphismes forts relatifs à la mise à jour des valeurs effectives des signaux ne peuvent être employés dans leur état actuel que si les variables relatives à la valeur effective, à l'attribut `EVENT` et à la valeur pilotée d'un même signal apparaissent dans cette ordre dans les DDD. Toutefois, il n'est pas nécessaire qu'elles soient contiguës. Cette limitation a été fixée pour des facilités de programmation. Toutefois, la levée de cette restriction ne pose aucun problème technique.

Les homomorphismes forts que nous avons défini sont pour la plus grande part construits sur les homomorphismes de base suivants :

`Hom selectCst(int var, int val)`

Homomorphisme sélectionnant les états pour lesquels la variable `var` a la valeur `val`.

`Hom setCst(int var, int val)`

Homomorphisme positionnant la valeur de la variable `var` à `val`.

`Hom selectExp(int var, const Expression& e)`

Homomorphisme sélectionnant les états pour lesquels la valeur de la variable `var` est égale à la valeur rendue par l'évaluation de l'expression `e`.

`Hom setExp(int var, const Expression& e)`

Homomorphisme positionnant la valeur de la variable `var` à celle rendue par l'évaluation de l'expression `e`.

La définition des deux premiers homomorphismes peut être trouvée dans le rapport du LaBRI. Les deux suivants sont une généralisation de l'homomorphisme réalisant l'affectation (ou la sélection conditionnelle) d'une variable par une autre variable. Cette généralisation est réalisée par le biais d'un gestionnaire d'expressions dont les fonctions de base sont :

- la construction d'expressions,
- l'identification de deux expressions,
- l'évaluation d'une expression en fonction de la valeur des variables impliquées.

Les différentes classes d'instructions composant le corps des processus, conduisent à la définition d'homomorphismes spécifiques.

3.3.1 Affectation

La forme générale d'une instruction d'affectation de variable est la suivante :

```
L1. v := exp;
L2. ...
```

Les identificateurs `L1` et `L2` désignent des valeurs constantes du compteur ordinal du processus (nommé `pc` par la suite). La simulation d'une telle instruction est réalisée par le biais de l'homomorphisme fort :

$$setCst(pc, L2) \circ setExp(v, exp) \circ selecCst(pc, L1)$$

Pour les états tels que le compteur ordinal désigne l'instruction en `L1`, la valeur de la variable `v` est affectée puis la valeur du compteur ordinal est positionnée à `L2`.

L'affectation de signal est traitée de manière similaire. Toutefois, la variable mémorisant le fait qu'une transaction a été réalisée dans le cycle d'exécution est positionnée en conséquence.

3.3.2 Branchement

La forme générale d'une instruction de branchement conditionnelle est la suivante :

```
L1. if cond then
L2.    ...
L3. else
L4.    ...
L5. end if;
```

L'homomorphisme correspondant à ce type d'instruction est le suivant :

$$setCst(pc, L2) \circ selExp(v, pc = L1 \wedge cond) +$$

$$setCst(pc, L4) \circ selExp(v, pc = L1 \wedge \neg cond)$$

Ce même schéma est appliqué a toutes les instructions de branchement conditionnel (`case`, `while`, `when ... next`, ...).

3.3.3 Suspension

La forme générale d'une telle instruction est :
`wait on s1,s2,..,sn until cond for exp;`

La prise en compte de ce type d'instruction est réalisée potentiellement en quatre étapes. En fonction de la situation (fin de delta cycle ou non), certaines étapes peuvent ne pas être nécessaires.

Lorsque le processus atteint l'instruction de suspension, son alarme est positionnée à la valeur de l'expression par le biais d'une affectation. Cette opération est réalisée par le biais de l'homomorphisme *setExp*.

A chaque fin de cycle d'exécution, les processus satisfaisant la condition de réveil sont sélectionnés. Cette sélection est réalisée par le biais d'un homomorphisme similaire à *selectExp* que nous avons spécialisé en vue d'en optimiser les performances.

Lorsqu'une fin de delta cycle est détectée, le temps réel est augmenté. Cela se traduit par une décrémentation des alarmes associées aux processus. La définition exacte de l'homomorphisme fort correspondant (*decAlarm*) est donnée dans l'annexe B. Il consiste en une recherche de la valeur minimale des alarmes suivie de la décrémentation de toutes les alarmes de cette valeur.

Enfin, l'exécution des instructions de suspension des processus réveillés provoque la mise à jour des valeurs effectives des signaux et de leurs attributs ainsi que celle des compteurs ordinaux. L'ensemble de ces opérations est réalisé par le biais des homomorphismes forts *setCst* et *setExp* ou des spécialisations de ceux-ci.

Une description détaillée de la structure de la bibliothèque (indiquant pour chaque classe, les méthodes principales les composants ainsi que pour chacune de ces méthodes les différents homomorphismes forts qui y sont invoqués) est donnée dans l'annexe B.

Chapitre 4

Bilan de l'étude

4.1 réalisation d'une bibliothèque VHDL-DDD

Le résultat concret de cette étude est la réalisation de la bibliothèque `VHDL-DDD`. Cette bibliothèque repose sur l'ensemble logiciel fourni par le LaBRI. Il est important de noter que notre bibliothèque supporte une classe très large de programmes VHDL et en particulier la manipulation de variables entières et la mise en oeuvre de contraintes temporelles (instruction de suspension avec une clause `for`). Dans son état actuel, `VHDL-DDD` permet le calcul de l'ensemble des états stables d'un programme ainsi qu'une fonction de sélection des états satisfaisant une expression quelconque. La bibliothèque a été employée pour la réalisation d'un outil de vérification de propriété de sûreté (i.e. relative à l'accessibilité).

Des premières mesures de performance ont été réalisées. Elles portent sur l'exemple de la section C.2 pour lequel le nombre de paquets et de retransmissions sont pris en tant que paramètres. Le tableau suivant récapitule les résultats obtenus.

Nb paquets	Nb répétitions	Nb états stables	Taille DDD	Taille arbre	Tps de calcul
1	5	96	306	2485	10.04 s
1	10	176	306	3605	15.98 s
10	5	12560	1195	205391	70.04 s
10	10	24560	1195	373391	80.21 s
20	5	49920	2045	769171	131.64 s
20	10	97920	2045	1.44e+06	144.40 s
30	5	112080	2895	1.69e+06	197.84 s
30	10	220080	2895	3.20e+06	213.16 s
40	5	199040	3745	2.97e+06	263.36 s
40	10	391040	3745	5.65e+06	279.94 s
50	5	310800	4595	4.61e+06	331.41 s
50	10	610800	4595	8.81e+06	351.32 s
60	5	447360	5445	6.60e+06	407.42 s
60	10	879360	5445	1.26e+07	433.35 s
70	5	608720	6295	8.96e+06	488.23 s
70	10	1.19e+06	6295	1.71e+07	510.94 s
80	5	794880	7145	1.16e+07	563.96 s
80	10	1.56e+06	7145	2.24e+07	588.29 s
30	20	436080	2895	6.22e+06	238.43 s
30	30	652080	2895	9.25e+06	269.29 s
30	40	868080	2895	1.22e+07	295.72 s
30	50	1.08e+06	2895	1.52e+07	319.65 s
30	60	1.30e+06	2895	1.83e+07	341.44 s
30	70	1.51e+06	2895	2.13e+07	366.93 s
30	80	1.73e+06	2895	2.43e+07	390.20 s
30	90	1.94e+06	2895	2.73e+07	412.32 s
30	100	2.16e+06	2895	3.04e+07	440,36 s

Ce tableau nous montre que, tout comme les diagrammes de décision binaire, un important partage peut être obtenu dans la représentation des états accessibles (plus de deux millions d'états peuvent être représentés par moins de trois milles cellules). Il est possible, ainsi, d'étudier l'algorithme pour un grand nombre de paquets, tout en obtenant des temps de calcul raisonnables. Surtout, nous voyons que, dans ce cas précis, l'augmentation du nombre de répétitions se fait à nombre de cellules constant et avec une petite augmentation du temps de calcul, alors que le nombre d'état augmente rapidement (plus de 200 000 états stables supplémentaires et 30 s de temps calcul de plus pour une augmentation de 10 répétitions avec un nombre de paquets égal à 30). Ces résultats sont très positifs et valident l'approche suivie dans cette étude.

L'évaluation de la bibliothèque doit maintenant se poursuivre et nous avons

contacté la société Thalès pour qu'elle nous fournisse des programmes VHDL plus réalistes.

Les perspectives que nous envisageons portent sur plusieurs voies. En premier lieu, les premières expérimentations que nous avons réalisées nous montrent que plusieurs optimisations de la bibliothèque DDD peuvent être envisagées. Elles portent sur l'ordonnancement des variables, les stratégies d'évaluations, les fonctions de hachage relatives au cache d'opération mais aussi sur la structure même des DDD et sur des extensions hiérarchiques. En second lieu, elles concernent la bibliothèque `VHDL-DDD`. En effet, nous pourrions élargir la part du langage supportée par la bibliothèque et y introduire les affectations différées, les signaux multi-pilotés (et les fonctions de résolution) ainsi que les appels de sous-programmes. Il est à noter que l'ensemble de ces points conduisent à un codage dynamique des états du système et que cette dynamicité peut tout à fait être prise en compte par les DDD. D'un point de vue performance, le gestionnaire d'expression actuellement employé nécessiterait d'être optimisé.

En dernier lieu et fort de notre expérience présente, nous pouvons aussi envisagé d'aborder d'autres domaines que la vérification de matériel. Un premier travail dans le domaine des réseaux de Petri a conduit à la réalisation d'un prototype présentant des performances très encourageantes. Le langage `promela`, proposé par l'outil `spin` pour la vérification de protocole de processus communicants par échanges de messages, par rendez-vous et par mémoire partagée, semble être un candidat tout désigné pour l'étude des systèmes parallèles.

4.2 Vérification modulaire

Le présent marché comprenait également une étude exploratoire des méthodes de vérification modulaire pour les systèmes décrits par BDD, et leur adaptation aux structures de type DDD. Les principaux résultats de cette étude sont détaillés dans le rapport du LIP6 daté d'octobre 2000 et rappelés dans cette section.

La vérification modulaire des systèmes matériels cherche à tirer partie de la structure de ces systèmes pour réduire le temps de vérification des propriétés. Nous nous plaçons maintenant dans le cadre de systèmes décrits sous forme d'une collection d'automates synchronisés, sur laquelle on cherche à vérifier un ensemble de propriétés CTL. L'idée majeure de cette vérification modulaire consiste à réduire chacun des automates avant de les synchroniser, puis à appliquer les algorithmes de vérification symboliques exposés précédemment.

La réduction est réalisée sur chaque composant en :

- masquant les informations de chaque automate qui ne nous sont pas nécessaires pour la vérification,
- calculant l'automate minimal équivalent à l'automate dont une partie des informations à été masquée.

La réduction de chaque composant sera d'autant plus importante que la quan-

tité d'informations masquée sera grande. C'est pour cette raison que nous effectuons une réduction de chaque composant *en fonction de chaque propriété à vérifier* : si notre système est composé de deux automates A1 et A2, et que l'on cherche à vérifier trois propriétés CTL P1, P2 et P3 sur le système global représenté par la synchronisation de A1 et A2, nous allons réduire A1 et A2 en fonction de P1, les synchroniser puis vérifier P1 sur leur produit, indépendamment, nous allons réduire A1 et A2 en fonction de P2, puis vérifier P2 sur leur produit et nous réaliserons le même traitement pour la vérification de P3, indépendamment des vérifications de P1 et P2.

Nous avons montré dans [REBM 98] que la réduction la plus grossière de chaque composant, lorsque l'on ne connaît pas a priori l'environnement dans lequel il est plongé, était obtenue en calculant la plus grande bisimulation incluse dans une relation d'équivalence masquant toutes les variables définissant un état d'un composant, à l'exception :

- des signaux de connexion reliant le composant à réduire aux autres composants avec lesquels il est synchronisé, et
- des variables de sortie intervenant dans les sous-formules propositionnelles de la propriété CTL à vérifier.

Plusieurs algorithmes ont été proposés pour calculer cet automate réduit. Citons pour mémoire [LN 91] et [Bouajjani 94]. Le premier est en fait un algorithme de minimisation d'une machine d'états finis (de type machine de Moore) déterministe et complète, représentée par une relation de transition sous forme de BDD, et permet d'obtenir directement un automate réduit. Le suivant s'applique à des automates plus généraux (notamment indéterministes et incomplets), et souffre d'un problème de représentation des données dans les étapes intermédiaires du calcul, ce qui le rend inapplicable dans notre cas, que nos systèmes soient représentés à l'aide de BDD et de relation de transitions booléenne, ou bien qu'ils soient représentés à l'aide de DDD et d'homomorphismes forts.

L'algorithme de réduction proposé dans [LN 91] nous semble le meilleur candidat pour les systèmes représentés par les techniques classiques de BDD, lorsque la relation de transition est représentée comme une fonction booléenne exprimant l'évolution de variables d'état futur en fonction de variables d'état courant. Le rapport d'octobre 2000 détaille les différentes étapes de l'algorithme de réduction, ainsi que les adaptations qui seraient nécessaires lorsque l'on considère des ensembles d'états décrits sous forme de VHDL-DDD, et que la relation de transition fait explicitement apparaitre deux jeux de variables : celui codant l'état courant et celui codant l'état futur. Pour mémoire, l'algorithme consiste à calculer les classes d'états équivalents par la relation de bisimulation et à construire un codage dense de ces classes d'équivalences, ainsi, on peut représenter le système sur un plus petit nombre de variables (booléennes). La relation de transition réduite est définie comme une composition de la fonction de codage dense et de la relation de transition initiale. En pratique, cette composition s'écrit à l'aide de quelques opérateurs booléens très simples, et la relation réduite représente

bien l'évolution du système (vis-à-vis de la propriété) sur un nombre de variable restreint, facilitant ainsi la vérification ultérieure.

Lorsque la relation de transition est représentée comme une composition d'homomorphismes forts, ce qui a été notre choix final, le principe de cet algorithme reste valable, mais la composition de la fonction de codage dense avec les homomorphismes forts doit être retravaillée : il faut construire un homomorphisme fort produisant les mêmes effets que la composition de l'homomorphisme fort initial et de la fonction de codage dense, en une seule étape, et travaillant directement sur le jeu de variable du modèle réduit. Cette étude est dans la lignée de nos perspectives générales concernant l'écriture des homomorphismes forts : A plusieurs reprises nous avons eu différentes possibilités de représentation d'un même traitement par un ou des homomorphismes (par exemple pour représenter l'augmentation du temps réel), ce qui avait une grande influence sur les performance d'évaluation de l'homomorphisme sur le DDD opérande.

4.3 Références

[LN 91] B. Lin, R. Newton, "Implicit Manipulation of Equivalence Classes using Binary Decision Diagrams", Int. Conf. on Computer-Aided Design, 1991.

[Bouajjani 94] A. Bouajjani, J.-C. Fernandez, N.Halbwachs, P. Raymond, C. Ratel, "Minimal State-Graph Generation", Sciences of Computer Programming, 1994.

[REBM 98] F. Rahim, E. Encrenaz, R. Bawa, M. Minoux, "Modular Model-checking of VLSI Designs described in VHDL", Int. Conf. On Computers and their Applications, 1998.

Annexe A

Manuel de référence

A.1 classe `Program`

`Program(const char* n="")`

Construit un programme de nom n.

A.2 classe `Attribut`

`operator Expression() const;`

Convertit un attribut en une expression.

A.3 classe `Signal`, `SignalIn`, `SignalOut`

A.3.1 Constructeurs

`Signal(Program& p, const char *n, int ini, int min, int max)`

Construit un signal dans le programme p. Ce signal porte le nom n, a comme valeur initiale ini et comme domaine de définition, l'intervalle $[min, max]$.

`SignalIn(Program& p, const char *n, int min, int max)`.

Construit un signal *d'entrée* dans le programme p. Ce signal porte le nom n et a comme domaine de définition, l'intervalle $[min, max]$.

`SignalOut(Program& p, const char *n, int min, int max)`

Construit un signal *de sortie* dans le programme p. Ce signal porte le nom n et a comme domaine de définition, l'intervalle $[min, max]$.

Un objet de type `SignalIn` ou `SignalOut` est accepté partout où un signal est attendu.

A.3.2 Méthodes

`operator Expression() const`

Convertit un signal en une expression.

`Attribut event() const`

Rend l'attribut event du signal.

`Attribut transaction() const`

Rend l'attribut transaction du signal.

A.4 classe `Process`

A.4.1 Constructeur

`Process(Program& p, const char* n, int pci)`

Contruit un processus du programme p portant le nom n et dont la valeur initiale du compteur ordinal est pci.

A.4.2 Méthode

`Expression al() const`

Rend l'expression correspondant à l'alarme du processus. Cette méthode ne devrait être employé qu'à des fins de vérification de propriétés.

`Expression pc() const`

Rend l'expression correspondant au compteur ordinal du processus. Cette méthode ne devrait être employé qu'à des fins de vérification de propriétés.

A.5 classe `Variable`

A.5.1 Constructeur

`Variable(Process& p, const char* n, int ini, int min, int max)`

Construit une variable dans le programme p. Cette variable porte le nom n, a comme valeur initiale ini et comme domaine de définition, l'intervalle $[min, max]$.

A.5.2 Méthodes

`operator Expression() const`

Convertit une variable en une expression.

A.6 classe Expression

A.6.1 Constructeur

`Expression(int i)`

Construit une expression constante de valeur i.

`Expression(const Expression& e)`

Construit une copie de l'expression e.

A.6.2 Opérateurs

`Expression& operator=(const Expression& e)`

Opérateur d'affectation.

`Expression operator+(const Expression& og, const Expression& od);`

Addition entre deux expressions.

`Expression operator-(const Expression& og, const Expression& od);`

Soustraction entre deux expressions.

`Expression operator-(const Expression& o)`

Négation arithmétique d'une expression.

`Expression operator*(const Expression& og, const Expression& od)`

Multiplication entre deux expressions.

`Expression operator/(const Expression& og, const Expression& od)`

Division entière entre deux expressions.

`Expression operator%(const Expression& og, const Expression& od)`

Reste de la division entière entre deux expressions.

`Expression operator==(const Expression& og, const Expression& od)`

Comparaison de deux expressions.

`Expression operator!(const Expression& o)`

Négation logique d'une expression.

`Expression operator<(const Expression& og, const Expression& od)`

Relation d'ordre entre deux expressions.

`Expression operator&&(const Expression& og, const Expression& od)`

Conjonction entre deux expressions.

`Expression operator||(const Expression& og, const Expression& od)`

Disjonction entre deux expressions.

Toutes les expressions prennent des valeurs entières. Une expression est considérée comme vraie si son évaluation rend une valeur différente de 0 et comme fausse dans le cas contraire. Les opérations logiques sont définies sur cette hypothèse.

A.7 fonctions et classes relatives aux instructions d'un processus

A.7.1 classe SensList

`SensList()`

Construit une liste de sensibilité vide.

`void insert(SensList& sl, const Signal& s)`

Ajoute l'attribut event du signal s à la liste de sensibilité sl.

`void insert(SensList& sl, const Attribut& a)`

Ajoute l'attribut a à la liste de sensibilité sl.

A.7.2 fonctions de déclaration d'instruction

```
void assignVar(Process& p, int cpc, int npc,
               const Variable& v, const Expression& e)
```

Ajoute l'instruction `v := e` au processus p. La valeur du compteur ordinal du processus correspondant à cette instruction est cpc et la valeur atteinte après exécution est npc.

```
void assignSig(Process& p, int cpc, int npc,
               const Signal& s, const Expression& e)
```

Ajoute l'instruction `s <= e` au processus p. La valeur du compteur ordinal du processus correspondant à cette instruction est cpc et la valeur atteinte après exécution est npc.

```
void branch(Process& p, int cpc, int npcif,
            int npcelse, const Expression& e)
```

Saut conditionnel. La valeur du compteur ordinal du processus correspondant à cette instruction est cpc. Si l'expression e est satisfaite alors la nouvelle valeur du compteur ordinal est $npcif$ et dans le cas contraire, elle est $npcelse$.

```
void assignAlarm(Process& p, int cpc, int npc,
                 const Expression& e)
```

Affecte à l'alarme du processus p la valeur de e. La valeur du compteur ordinal du processus correspondant à cette instruction est cpc et la valeur atteinte après exécution est npc.

```
void wait(Process& p, int cpc, int npc,
          const SensList& sl, const Expression& e, bool b)
```

Instruction de suspension. La valeur du compteur ordinal du processus correspondant à cette instruction est cpc et la valeur atteinte après exécution est npc. Le processus attend sur les événements et transactions contenus dans sl et jusqu'à ce que l'expression e soit satisfaite. Si un délai de réveil est spécifié, le paramètre b doit être positionné à vrai.

A.8 classe Simulator

`Simulator(const Program& p)`

Construit un simulateur du programme p.

`DDD reachable() const`

Renvoie le DDD représentant l'ensemble des états stables du programme simulé.

A.9 fonction select

`DDD select(const DDD& d, const Expression& e)`

Rend le sous-ensemble des états de d satisfaisant l'expression e.

Annexe B

Structuration interne de la bibliothèque VHDL-DDD

B.1 Classes, méthodes et invocations

La figure B.1 décrit la structure interne des classes composant la bibliothèque. Pour chaque classe est indiqué le nom des méthodes impliquées dans la simulation et pour chacune d'entre elles sont spécifiés les appels de fonctions ou de méthodes qui y sont réalisés. On peut remarquer que le cycle de simulation décrit dans la section 3.2 a une traduction immédiate dans la structure interne de la bibliothèque.

B.2 Incrémentation du temps réel

L'homomorphisme fort réalisant la mise à jour des alarmes des processus lors du traitement d'un état stable est défini comme suit.

$$decAlarm(A) = cut() \circ d^{\downarrow}(A, 0)$$

où le paramètre A représente l'ensemble des alarmes des processus.

L'homomorphisme $d^{\downarrow}$ recherche la valeur minimale non nulle des alarmes. Le deuxième paramètre de cet homomorphisme correspond à la valeur minimal connue à l'instant de l'appel. La valeur minimale trouvée est stockée en amont du DDD construit pour permettre les décrémentations successives. Ce préfixe est supprimé du résultat final par l'homomorphisme cut. Les décrémentations successives sont réalisées par l'homomorphisme fort $d^{\uparrow}$.

$$\begin{array}{lcl} d^{\downarrow}(A, min)(e, x) & = & \begin{cases} d^{\uparrow}(true, a, x) \circ d^{\downarrow}(A, x) & \text{si} \quad e \in A \wedge x > 0 \wedge \\ & & (min > x \vee min = 0) \\ d^{\uparrow}(false, a, x) \circ d^{\downarrow}(A, min) & \text{sinon} \end{cases} \\ d^{\downarrow}(A, min)(1) & = & 0 \xrightarrow{min} 1 \end{array}$$

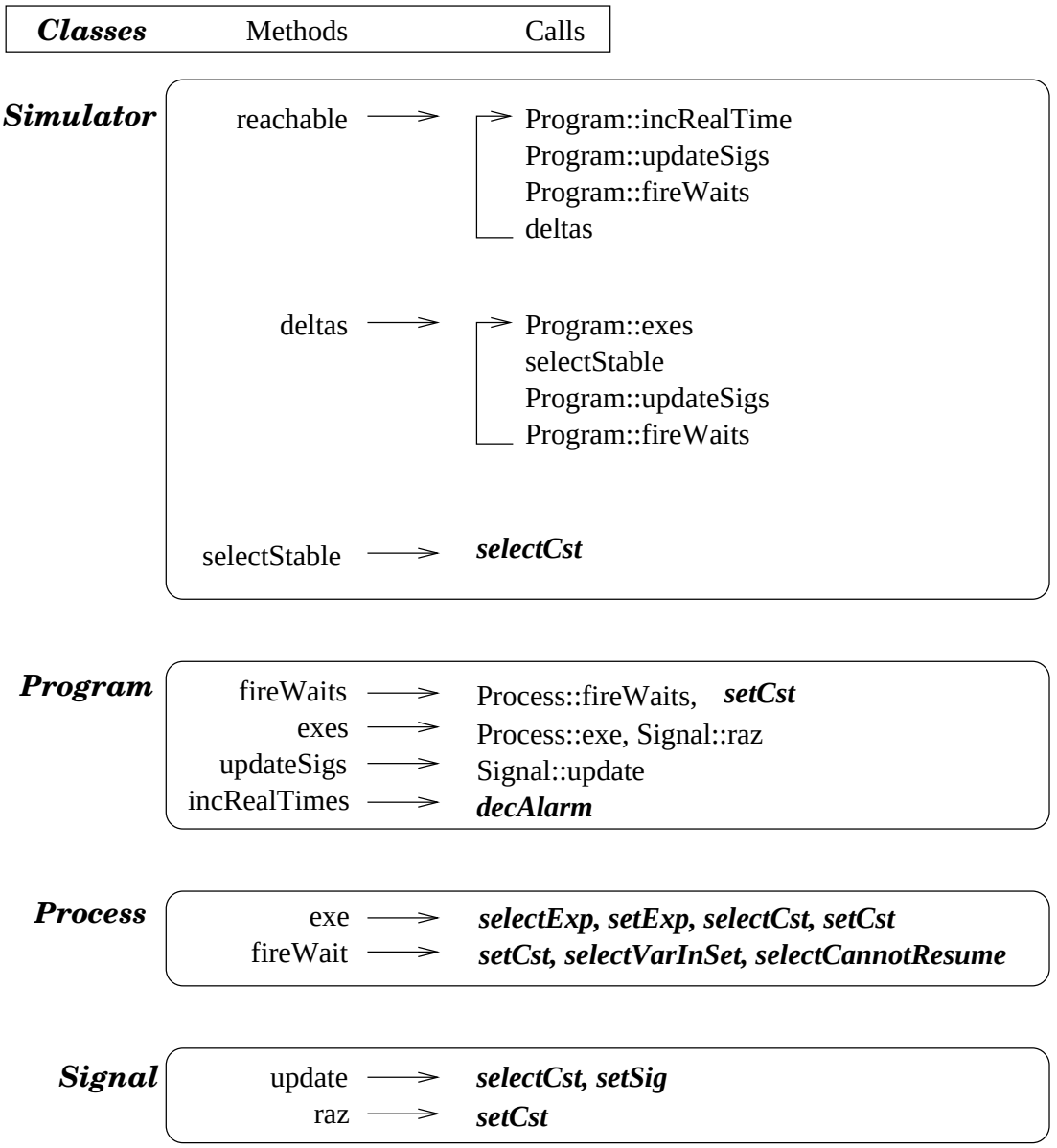

FIG. B.1 – Structure interne du programme

$$
\begin{array}{lcl}
d^{\uparrow}(b, var, val)(e, x) & = & \left\{ \begin{array}{ll} e \xrightarrow{x} var \xrightarrow{val-x} Id & \text{si } b \wedge val > 0 \\ e \xrightarrow{x} var \xrightarrow{val} Id & \text{sinon} \end{array} \right. \\
d^{\uparrow}(b, var, val)(1) & = & \top
\end{array}
$$

$$
\begin{array}{lcl}
cut()(e, x) & = & Id \\
cut()(1) & = & \top
\end{array}
$$

B.3 `VerifVHDL`

L'outil `VerifVHDL` supporte un sous-ensemble du langage VHDL. Ce sous-ensemble est décrit par la grammaire suivante :

```
Program ::= ListeVideCR Declarations1 DeclEntity DeclArchi CorpsProgram

DeclEntity ::= ENTITY IDENT IS ListeVideCR PORT PARENTH_O Declarations SignauxExt1
                                    PARENTH_F ListeVideCR END IDENT PV ListeVideCR

DeclArchi ::= ARCHITECTURE IDENT OF IDENT IS

CorpsProgram ::= Declarations Signaux1 DeclConstante DEBUT ListeProcess FinProgram

SignauxExt ::= /* epsilon */ | ListeCR SignauxExt1

SignauxExt1 ::= /* epsilon */ | SyntSignauxExt SignauxExt

SyntSignauxExt ::= ListeIdent DEUXPTS TypeExt IDENT PV
                 | ListeIdent DEUXPTS TypeExt INTEGER RANGE NOMBRE TO NOMBRE PV
                 | ListeIdent DEUXPTS TypeExt INTEGER PV
                 | ListeIdent DEUXPTS TypeExt TIME PV
                 | ListeIdent DEUXPTS TypeExt BIT PV

TypeExt ::= IN | OUT

Declarations ::= /* epsilon */ | ListeCR Declarations1

Declarations1 ::= /* epsilon */
                | TYPE IDENT IS INTEGER RANGE NOMBRE TO NOMBRE PV Declarations
                | TYPE IDENT IS PARENTH_O ListeIdent PARENTH_F PV Declarations
Signaux ::= /* epsilon */ | ListeCR Signaux1

Signaux1 ::= /* epsilon */ | SyntSignaux Signaux

SyntSignaux ::= SIGNAL ListeIdent DEUXPTS IDENT AFFECT NOMBRE PV
              | SIGNAL ListeIdent DEUXPTS IDENT AFFECT IDENT PV
              | SIGNAL ListeIdent DEUXPTS INTEGER RANGE NOMBRE TO NOMBRE AFFECT NOMBRE PV
              | SIGNAL ListeIdent DEUXPTS INTEGER AFFECT NOMBRE PV
              | SIGNAL ListeIdent DEUXPTS TIME AFFECT NOMBRE MS PV
              | SIGNAL ListeIdent DEUXPTS BIT AFFECT NOMBRE PV

ListeProcess ::= /* epsilon */ | ListeCR Process

Process ::=  /* epsilon */
           | IDENT DEUXPTS PROCESS DeclVariables DEBUT ListeInstructions END
                                                PROCESS IDENT PV ListeProcess

DeclConstantes ::= /* epsilon */ | ListeCR DeclConstante

DeclConstante ::= /* epsilon */ | SyntDeclVar DeclConstantes

SyntDeclVar ::= CONSTANTE ListeIdent DEUXPTS IDENT AFFECT NOMBRE PV
              | CONSTANTE ListeIdent DEUXPTS IDENT AFFECT IDENT PV
              | CONSTANTE ListeIdent DEUXPTS INTEGER RANGE NOMBRE TO NOMBRE AFFECT NOMBRE PV
              | CONSTANTE ListeIdent DEUXPTS INTEGER AFFECT NOMBRE PV
              | CONSTANTE ListeIdent DEUXPTS TIME AFFECT NOMBRE MS PV
              | CONSTANTE ListeIdent DEUXPTS BIT AFFECT NOMBRE PV

DeclVariables ::= /* epsilon */ | ListeCR DeclVariable

DeclVariable ::=  /* epsilon */ | SyntDeclVar DeclVariables

SyntDeclVar ::= VARIABLE ListeIdent DEUXPTS IDENT AFFECT NOMBRE PV
              | VARIABLE ListeIdent DEUXPTS IDENT AFFECT IDENT PV
              | VARIABLE ListeIdent DEUXPTS INTEGER RANGE NOMBRE TO NOMBRE AFFECT NOMBRE PV
              | VARIABLE ListeIdent DEUXPTS INTEGER AFFECT NOMBRE PV
              | VARIABLE ListeIdent DEUXPTS INTEGER PV
              | VARIABLE ListeIdent DEUXPTS TIME AFFECT NOMBRE MS PV
              | VARIABLE ListeIdent DEUXPTS BIT AFFECT NOMBRE PV
```

```
ListeInstructions ::= /* epsilon */ | ListeCR Instruction

Instruction ::= /* epsilon */ | SyntInstruction ListeInstructions

SyntInstruction ::= IDENT INFEGAL Expression PV
                  | IDENT AFFECT Expression PV
                  | WAIT PV
                  | WAIT ON ListeIdentWait PV
                  | WAIT FOR Expression MS PV
                  | WAIT ON ListeIdentWait FOR Expression MS PV
                  | WAIT UNTIL Expression PV
                  | WAIT ON ListeIdentWait UNTIL Expression PV
                  | WAIT UNTIL Expression FOR Expression MS PV
                  | WAIT ON ListeIdentWait UNTIL Expression FOR Expression MS PV
                  | IDENT DEUXPTS WHILE Expression LOOP ListeInstructions END LOOP PV
                  | NEXT PV
                  | EXIT IDENT PV
                  | IF Expression THEN ListeInstructions SuiteIf

SuiteIf ::= END IF PV | ELSE ListeInstructions END IF PV

Expression ::= Operande
             | PARENTH_O Expression PARENTH_F
             | MOINS Expression
             | NOT Expression
             | Expression PLUS Expression
             | Expression MOINS Expression
             | Expression MULTIPLIE Expression
             | Expression MOD Expression
             | Expression DIV Expression
             | Expression EGAL Expression
             | Expression INF Expression
             | Expression SUP Expression
             | Expression DIFFERENT Expression
             | Expression SUPEGAL Expression
             | Expression INFEGAL Expression
             | Expression AND Expression
             | Expression OR Expression

Operande ::= IDENT | NOMBRE | IDENT TICK EVENT

FinProgram ::= END IDENT PV ListeVideCR

ListeIdent ::= IDENT | IDENT VIRGULE ListeIdent

ListeIdentWait ::= SyntIdentWait | SyntIdentWait VIRGULE ListeIdentWait

SyntIdentWait ::= IDENT | IDENT TICK EVENT | IDENT TICK TRANSACTION

ListeVideCR ::= /* epsilon */ | ListeCR

ListeCR ::= CR | CR ListeCR
          | MOINS MOINS Commentaire ListeCR
```

Annexe C

Exemples d'utilisation de la bibliothèque

C.1 Exemples simples

C.1.1 Instruction `while`

Source VHDL

```
architecture A of E is

constant MAX: integer := 15;

begin

P1 : process
        variable i : integer := 0;
        begin
        while i<MAX loop
                i := i + 1;
                wait for 1 ns;
        end loop;
        i := 0;
end process P1;

end A;
```

Source C++ correspondant

```
#include "DDD.h"
#include "vhdl.hpp"
#include "simulator.hpp"
```

```
int main() {
        const int MAX = 15;
        enum { L11=11,L12,L13,L14,L15};

        Program p;

        Process p1(p, "p1", L11);
                Variable i(p1, "i", 0, −32000, 32000);
                branch          (p1, L11, L12, L15, i < MAX);
                assignVar       (p1, L12, L13, i , i+1);
                assignAlarm     (p1, L13, L14, 1);
                wait            (p1, L14, L11, SensList (), true, true);
                assignVar       (p1, L15, L11, i , 0);

        Simulator sim(p);

        double d = getTime();
        DDD reach = sim.reachable();
        double tps = getTime() − d;

        cout << "size of the set : " << reach.nbStates() << endl;
        cout << "size of the DDD: " << reach.size() << endl;
        cout << "computation time: " << tps << "s." << endl;

        return 0;
}
```

C.1.2 Signaux et instruction `wait`

Source VHDL

```
architecture A of E is

constant MAX: integer := 15;
signal S1, S2 : integer range 0..MAX := 0;

begin

P1 : process
        variable V1 : integer := 1;
        begin
        S1 <= V1;
        wait on S1,S2 until S1/=0 or S2/=0 and V1/=0 for 10 ns;
        S1 <= (S1 + 1) mod MAX;
        V1 := S2;
        wait for 15 ns;
```

```
end process P1;

P2 : process
        begin
        S2 <= (MAX+(S2 - 1) mod MAX) mod MAX;
        wait on S1 for 5 ns;
        wait on S2 for 10 ns ;
        S2 <= (S2 + 2) mod MAX;
        wait on S1;
end process P2;

end A;
```

Source C++ correspondant

```
#include "DDD.h"
#include "vhdl.hpp"
#include "simulator.hpp"

int main() {

        const int MAX = 15;
        enum { L11=11,L12,L13,L14,L15,L16,L17,
               L21=21,L22,L23,L24,L25,L26,L27};

        Program p;

        Signal  s1(p, "S1" , 0, 0,  MAX),
                s2(p, "S2" , 0, 0,  MAX);

        Process p1(p, "P1", L11);

                Variable V1(p1, "v1", 1, MININT, MAXINT);
                SensList sl12;
                insert(sl12, s1.event());
                insert(sl12, s2.event());

                assignSig       (p1, L11, L12, s1, v1);
                assignAlarm     (p1, L12, L13, 10);
                wait            (p1, L13, L14, sl12, s1 || s2 && v1, true);
                assignSig       (p1, L14, L15, s1, (s1+1)%MAX);
                assignVar       (p1, L15, L16, v1, s2);
                assignAlarm     (p1, L16, L17, 15);
                wait            (p1, L17, L11, SensList(), true, true);

        Process p2(p, "P2", L21);
```

```
                SensList sl1;
                insert(sl1, s1.event());
                SensList sl2;
                insert(sl2, s2.event());

                assignSig      (p2, L21, L22, s2, (MAX+(s2-1)%MAX)%MAX);
                assignAlarm    (p2, L22, L23, 5);
                wait           (p2, L23, L24, sl1, true, true);
                assignAlarm    (p2, L24, L25, 10);
                wait           (p2, L25, L26, sl2, true, true);
                assignSig      (p2, L26, L27, s2, (s2+2)%MAX);
                wait           (p2, L27, L21, sl1, true, false);

        Simulator sim(p);

        double d = getTime();
        DDD reach = sim.reachable();
        double tps = getTime() - d;

        cout << "size of the set: " << reach.nbStates() << endl;
        cout << "size of the DDD: " << reach.size() << endl;
        cout << "computation time: " << tps << "s." << endl;

        return 0;
}
```

C.2 Un protocole de communication avec pertes

Seul le code VHDL du protocole est donné. Cet exemple illustre comment peuvent être spécifiées des propriétés d'états (clause `assert`) et de séquences (processus concurrent et clause `assert`).

C.2.1 Bibliothèque de types

```
library les_types is

-- Nombre maximal de paquets : un parametre du modele
constant MAX_MESS : integer := 10;

-- Nombre maximal de repetitions : un parametre du modele
constant nr_lim : integer := 3;

type MESS is integer range 1 to MAX_MESS;
```

```
type type_fichier   is record
                            rq  :  bit;
                            nb_msg : MESS;
                       end record;
type type_conf is (OK,NOK,DTKNW);

type IND is record
                msg : integer;
                etat  : ( FIRST,LAST,INCOMPLETE);
            end record;

end les_types;
```

C.2.2 Protocole

```
library les_types;                  -- contient les types du port
use les_types.all;


entity BRP is
port    (
    REQ : in type_fichier;   -- record avec rq (bit) et nb_msg a
                             -- transmettre (MESS)
    CONF   : out type_conf;      -- OK, NOK, DTKNW
    IND : out type_ind;      -- record avec msg (la donnee), et
                    -- l'etat du paquet =FIRST,LAST,INCOMPLETE
    IND_ERR : out bit        -- YES, NO
    )
end BRP;

architecture BRP_1 of BRP is

type CANAL_M is (MSG_FIRST, MSG_INCMP,MSG_LAST,ACK,VIDE);
type CANAL_T is bit;

type TYPE_CANAL is record
    message : CANAL_M;
    toggle   : CANAL_T;
end record;

signal K_in, K_out : type_canal;
signal L_in, L_out : type_canal;

begin

sender   : process
```

```
                variable nb_msg : integer;
            variable nb, nr : integer;
            variable envoi  : type_canal;
            variable tg : CANAL_T;
            constant timer1 : time := 100 ms;

        begin
            wait on REQ until REQ.rq = 1;
            nb_msg := REQ.nb;
            nb := 0;

transmission:   while not (nb > nb_msg) loop
                nr := 0; nb := nb + 1;
                if (nb < nb_msg) and (nb = 1)) then
                    envoi.message := MSG_FIRST;
                elsif nb < nb_msg then
                    envoi.message := MSG_INCMP;
                else
                    envoi.message := MSG_LAST;
                end if;
                envoi.toggle := tg;
                K_in <= envoi;

                wait on L_out until (L_out.message = ACK) for timer1;

                if ((L_out'event) and (nb = nb_msg)) then
                    -- succes transfert du fichier complet
                    CONF <= OK;
                    tg := not tg;
                    exit;
                elsif ((L_out'event) and (nb < nb_msg)) then
                    -- succes transfert un message
                    tg := not tg;
                    next;
                    else -- wait franchi sur timer1 : pb de transmission du msg
                    -- essai de retransmission du meme message
retransmission:         while (nr < nr_lim) loop
                        nr := nr + 1;
                        K_in <= envoi;

                        wait on L_out until (L_out.message = ACK) for timer1;

                        if (L_out'event) then
                            -- succes transfert un message
                            tg := not tg;
```

```
                    exit; -- sort de la boucle retransmission
                elsif (nr = nr_lim) then
                    -- echec transfert un message
                    if (nb = nb_msg) then
                        CONF <= DTKNW;
                    else
                        CONF <= NOK;
                    end if;
                    exit transmission; -- sort des 2 boucles

                else -- essai de retransmission
                    next;
                end if;
            end loop;
        end if;
    end loop;
end process sender;

channel_K : process
    begin
    wait on K_in;
    if perte_K then
        K_out.message <= VIDE;
    else
        K_out <= K_in;
    end if;
end process channel_K;


receiver : process
    variable envoi : TYPE_CANAL;
    variable tg    : CANAL_T;
    constant timer2 : time := 20 ms;

   begin
    wait on K_out;
    IND <= K_out.message;
    IND_ERR <= 0;
    tg := K_out.toggle;
    envoi.message := ACK;
    envoi.toggle := tg;
    L_in <= envoi;

    wait on K_out until not (K_out.message = VIDE) for timer2;
```

```
        while (K_out'event) loop
            if (K_out.toggle = tg) then
                -- ack non recu : renvoi du meme toggle
                envoi.message := ACK;
                envoi.toggle := tg;
                L_in <= envoi;
            else
                -- ack recu : traitement du msg suivant
                tg := K_out.toggle;
                envoi.message := ACK;
                envoi.toggle := tg;
                L_in <= envoi;
                if (K_out.message = MSG_LAST) then
                    exit;
            end if;
            wait on K_out until not (K_out.message = VIDE) for timer2;
        end loop;
        if not(K_out'event) then
            IND_ERR <= 1;
        end if;
    end process receiver;

channel_L : process
        wait on L_in;
        if perte_L then
            L_out.message <= VIDE;
        else
            L_out <= L_in;
        end if;
    end process channel_L;

-------------------------------------
-- expression de propriétés

-- les canaux K et L ne transmettent pas de message simultanément

        property_1 : assert(not (K_in.message /= VIDE) or L_in.message = VIDE);

-- si un message est convenablement transmis et acquitté, il n'est pas
-- réémis (le toggle change)
            property_2 : process
                        variable svg_tg : CANAL_T;
            begin
                wait on K_in until K_in.message /= VIDE;
                svg_tg := K_in.toggle;
```

```
            loop
            wait on L_out until L_out.message /= VIDE and L_out.toggle = svg_tg;
            wait on K_in until K_in.message /= VIDE;
            if K_in.toggle = svg_tg then
                erreur : wait;
            else
                svg_tg := K_in.toggle;
            end if;
            end loop;
        end process;
        property_2b : assert (property_2.pc = property_2.erreur);

end BRP_1;
```