Université de Montréal

UN OUTIL POUR LA SPÉCIFICATION DE MATÉRIEL ET LA GÉNÉRATION DE MODÈLES EXÉCUTABLES

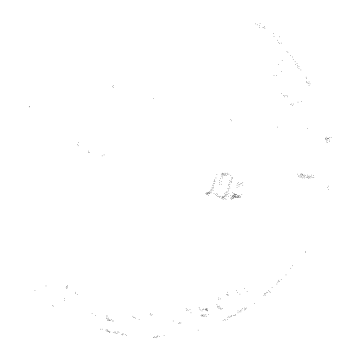
par

Philippe-André Babkine

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

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

mars 1996



Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé

UN OUTIL POUR LA SPÉCIFICATION DE MATÉRIEL ET LA GÉNÉRATION DE MODÈLES EXÉCUTABLES

présenté par

Philippe-André Babkine

a été évalué par un jury composé des personnes suivantes :

El Mostapha Aboulhamid

(président-rapporteur)

Eduard Cerny

(directeur de recherche)

François Lustman

(membre du jury)

Mémoire accepté le :

4 juillet 1996

SOMMAIRE

Ce mémoire présente un outil et une méthodologie permettant le développement rapide de modèles structurés dans le but de vérifier la fonctionnalité de systèmes numériques VLSI.

Le modèle d'un système numérique s'élabore en décrivant le comportement du système tel qu'observé à son interface avec le monde extérieur. Les chronogrammes représentent les opérations de base de l'interface du système, sur lesquelles sont mises en évidence les relations temporelles entre les événements composant l'opération. Ceci permet de décrire précisément l'aspect temporel (*timing*) de l'opération. Son aspect fonctionnel est capturé en annotant de procédures et de fonctions, les événements d'un chronogramme.

La composition à l'aide d'opérateurs hiérarchiques des diagrammes chronologiques de base complète la description du comportement du système tel qu'observable à son interface. Idéalement, les diagrammes chronologiques hiérarchiques annotés sont capturés graphiquement. Un langage de description de ces diagrammes est proposé afin de servir de forme intermédiaire entre les outils de capture et des applications qui utilisent les diagrammes.

Une de ces applications est traitée: soit la génération de modèles exécutables de système en langage de description de matériel, à partir de leur description sous forme de chronogrammes hiérarchiques annotés. L'aspect algorithmique de cette simulation est abordé et un exemple précisant les concepts de modélisation est développé.

La modélisation, en composant hiérarchiquement les opérations de base d'un système sur lesquelles l'aspect temporel et fonctionnel est capturé, permet le développement rapide et précis de modèles. La possibilité d'exécuter ces modèles permet de vérifier le comportement d'un système avant même de commencer le développement de sa réalisation matérielle. Les problèmes potentiels du système sont ainsi identifiés à un stade où il est moins coûteux d'y remédier.

Mots clés : VLSI, spécification, vérification, simulation, modèle exécutable.

TABLE DES MATIÈRES

LISTE DES TABLES

LISTE DES FIGURES

CHAPITRE 1

Introduction

Le développement de matériel numérique VLSI est un processus complexe dont les grandes étapes s'étendent de l'idée originale d'un produit jusqu'à la réalisation matérielle. Entre ces deux extrêmes, plusieurs étapes de conception se succèdent.

L'idée originale mène à la spécification du design dont la forme la plus courante est celle d'un document décrivant les détails des réalisations matérielles. Typiquement, il y est décrit comment interagissent les composants VLSI des réalisations. Souvent, ceux-ci sont des sous-systèmes dont les détails de réalisation restent à définir ou des composants numériques disponibles sur le marché. Dans tous les cas, la description qui en est faite détaille leur interface sans qu'il ne soit nécessaire de connaître la réalisation interne du composant.

L'interface d'un système se situe à la frontière entre ce dernier et l'environnement extérieur. Toute interaction entre le système et l'environnement peut donc être observée à l'interface. Par ailleurs, la description de l'interface d'un système VLSI se fait à plusieurs niveaux d'abstraction. A un niveau, il est possible de décrire l'apparence physique de l'interface du système alors qu'à un niveau plus élevé, il convient de parler de la forme des échanges d'information entre le système et l'environnement. Comme il est ici question de modélisation de systèmes numériques VLSI, le niveau d'abstraction de l'interface se situe au niveau logique booléen ou plus haut.

L'interface (au niveau logique) d'un composant est décrite en considérant deux aspects complémentaires:

(i) la fonctionnalité du composant

(ii) la description de son aspect temporel

Le premier aspect prend généralement la forme d'une description textuelle tandis que le deuxième est présenté le plus souvent sous forme de chronogrammes. La figure 1.1 en est un exemple. Dans la figure, les signaux de l'interface d'un système sont disposés verticalement. Les événements (changement de valeur des signaux) sont illustrés horizontalement. Certains

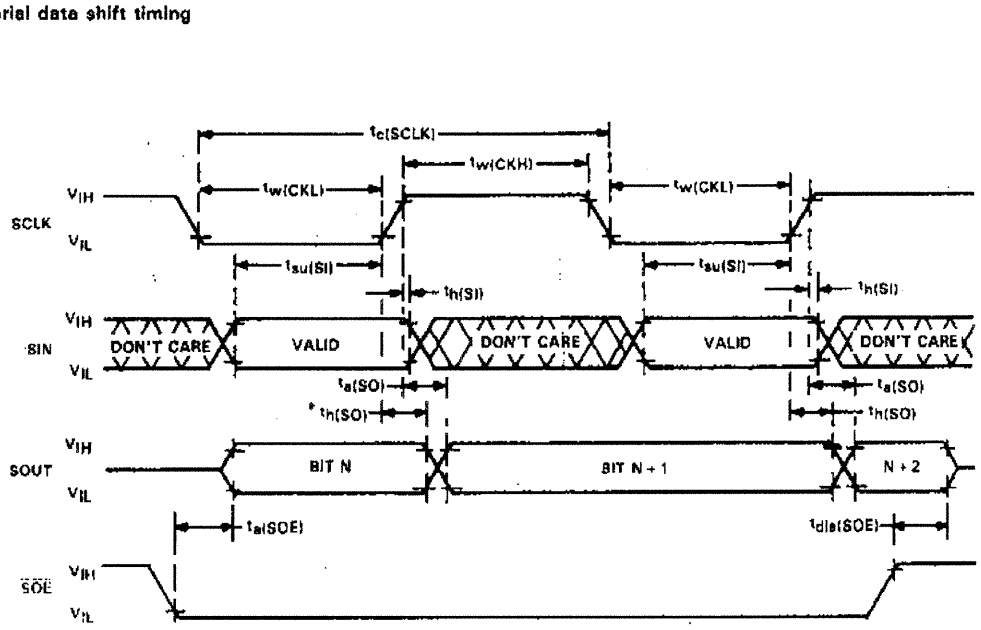


FIGURE 1.1. Diagramme chronologique typique.

événements sont mis en relation temporelle et ce fait est traduit graphiquement par les flèches entre les événements.

L'aspect temporel dans sa généralité décrit les relations temporelles régissant le fonctionnement du composant. L'aspect temporel de l'interface, plus spécifiquement, décrit les relations temporelles entre les événements observables à l'interface du système.

1. DESCRIPTION DE COMPOSANTS

Une description textuelle de l'aspect fonctionnel d'un composant est une description informelle se prêtant mal à un usage autre que la documentation du design. Si leur forme le permettait, la simulation et la vérification formelle de systèmes seraient d'autres usages possibles de ces descriptions.

Les modèles exécutables sont une autre forme possible de descriptions. Ils sont écrits dans un langage de programmation et constituent ainsi une description formelle. Ils sont dits exécutables, parce qu'après compilation, l'exécution du programme résultant sur un ordinateur simulera le fonctionnement ou certains aspects du comportement du composant.

L'aspect temporel de l'interface d'un composant est difficilement intégrable aux modèles exécutables, lui qui était si clair lorsque présenté sous forme de chronogramme. La nature des langages de description de matériel fait en sorte qu'il n'est pas simple d'intégrer les contraintes temporelles aux modèles exécutables.

Il est question dans ce mémoire d'un outil et d'une méthodologie permettant de spécifier l'aspect temporel et fonctionnel d'un composant ainsi que d'un des usages de ces spécifications: la génération de modèles exécutables. Ces travaux donnent suite au modèle de simulation basé sur les graphes de contraintes hiérarchiques proposé par [Duf91].

La méthodologie de spécification d'un système consiste à exprimer le comportement du système tel qu'il est observable à l'interface de celui-ci. Les opérations de base du système sont décrites par des chronogrammes feuilles consistant simplement en des chronogrammes (voir figure 1.1) annotés fonctionnellement de procédures et de fonctions. Les chronogrammes feuilles sont combinés hiérarchiquement grâce à des opérateurs pour décrire des comportements plus complexes. Ces descriptions portent le nom de *chronogrammes hiérarchiques annotés.*

D'un chronogramme hiérarchique annoté se dérive un modèle exécutable du système dans le langage de description VHDL. Le modèle, lorsque exécuté, valide le temps d'occurrence des événements générés par l'univers du système en fonction des contraintes temporelles et des valeurs des événements spécifiées dans la description de l'interface du système. Le modèle produit aussi les événements de sortie du système aux moments et aux valeurs dictés encore une fois par la description de l'interface du système. Le modèle d'un système peut être exercé pour vérifier son comportement suite à certains stimuli provenant de son environnement.

D'autres approches à la vérification existent mais celles basées sur la simulation sont les plus populaires dans l'industrie. Les travaux sur la vérification basée sur la simulation présentés dans ce mémoire constituent une partie d'un effort de recherche plus important sur la vérification de matériels numériques. D'autres méthodes de vérification plus formelles sont au stade de la recherche. Ces travaux portent sur les items suivants:

- la vérification de compatibilité temporelle de système
- la synthèse logique de système à partir de spécifications

Tous ces efforts de recherche partagent un formalisme de description des systèmes basé sur les *chronogrammes hiérarchiques annotés.*

2. Revue de littérature

La recherche en vérification de matériel de système est un domaine vaste qui intéresse beaucoup de chercheurs. Les méthodes de vérification basées sur des descriptions d'interfaces

sous forme de chronogrammes sont introduites par [Bor88]. L'auteur démontre l'utilité des chronogrammes en tant que constituants de la spécification de l'interface de systèmes numériques et propose une méthode de synthèse de blocs traducteurs entre les interfaces de deux systèmes à priori incompatibles. Les interfaces sont capturées à l'aide de chronogrammes. Le bloc traducteur est la réalisation logique nécessaire pour compléter l'interface de deux systèmes entre eux et la réalisation ainsi obtenue devient valide parce qu'elle est atteinte par un processus formel.

Une autre approche à la vérification basée sur les chronogrammes est présentée dans [ALG91] où l'approche proposée consiste à s'assurer qu'un système possède certaines propriétés qui prouvent l'exactitude du système. Celui-ci est modélisé par une structure de Kripke (automate fini sur les états duquel est associé un ensemble de propriétés exprimées dans une logique temporelle). Les chronogrammes, quant à eux, expriment les propriétés à vérifier. Ceux-ci sont ensuite traduits méthodiquement en propriétés sur les structures de Kripke. La vérification consiste à s'assurer que la structure de Kripke possède les propriétés exprimées par les chronogrammes. Il existe des algorithmes capables de vérifier qu'une structure possède les propriétés exprimées. Des réserves sont à considérer quand à l'éventail des propriétés exprimables par les chronogrammes. De plus cette méthode suppose que l'on dispose d'un modèle de Kripke du système à vérifier.

Une approche similaire se retrouve dans [SD93] où les chronogrammes décrivent qualitativement l'ordonnancement des événements. Ces ordres partiels sont représentés par des graphes de simulation dans lesquels il est possible d'extraire toutes les séquences d'exécutions d'événements possibles d'un chronogramme. Une procédure de transformation s'applique à transformer les graphes de simulation en formule de la logique temporelle PTL. Quand sont combinés les différents chronogrammes d'une réalisation, il en résulte une formule PTL pour laquelle il existe un algorithme efficace (polynômial) qui détermine la compatibilité des chronogrammes entre eux.

3. Contributions

Les contributions originales de ce travail sont :

- La réalisation de l'outil de capture graphique de chronogrammes feuilles. Cet outil permet l'édition graphique de chronogrammes semblables à celui présenté à la figure 1.1.
- L élaboration d'un langage intermédiaire entre la forme graphique d'un chronogramme et des outils acceptant en entrée des chronogrammes hiérarchiques annotés.
- La réalisation d'un outil de simulation de chronogrammes hiérarchiques annotés, basé sur les travaux de [Duf91]. L'outil accepte en entrée un chronogramme hiérarchique

annoté puis à partir de cette description d'interface de système, l'outil génère un modèle exécutable du système dans le langage de description de matériel VHDL. La réalisation de l'outil de simulation est basé sur la réalisation proposée dans [Duf91] dans laquelle une partie de l'algorithme de simulation a été modifié pour être plus efficace. De plus, le traitement de la partie fonctionnelle d'un modèle a été étendue. L'algorithme traite maintenant un plus grand nombre *d'annotations*.

4. Plan du mémoire

Ce mémoire se divise comme ceci: le chapitre 2 présente le concept de chronogrammes hiérarchiques fonctionnels servant à spécifier les systèmes numériques. Le concept de simulation de chronogrammes hiérarchiques annotés est aussi traité dans ce chapitre. Le chapitre 3 est un survol des sous-systèmes composant l'outil de spécification et simulation. Le chapitre 4 est un exemple d'application de la méthodologie; un système numérique simple y est décrit. La méthodologie de spécification ainsi que l'usage de l'outil est démontré dans cet exemple. Le chapitre 5 reprend chacun des sous-systèmes et présente les caractéristiques les plus importantes de leur réalisation. Finalement, la conclusion de ce mémoire résume les travaux. Une description en forme normale de Backus du langage de représentation des chronogrammes hiérarchiques annotés et les références bibliographiques se trouvent en annexe de ce mémoire, de même que des sections du code nécessaire à la simulation de chronogramme.

CHAPITRE 2

Les chronogrammes hiérarchiques annotés

Les chronogrammes hiérarchiques annotés servent à décrire des systèmes numériques. L'éventail des descriptions réalisables grâce aux chronogrammes va du simple composant VLSI discret (une puce) jusqu'à de l'équipement complexe de traitement de l'information (comme un commutateur téléphonique). Décrire le comportement d'un système tel que vu de son interface avec l'extérieur consiste à décrire et annoter les séquences d'événements perçues et les réponses du système.

Une description d'un système faite du point de vue de son interface est composée à partir des descriptions des opérations de base de l'interface du système. Par exemple, les opérations de base d'un boîtier de mémoire sont la lecture et l'écriture d'informations dans la mémoire. Les opérations de base sont décrites par des chronogrammes feuilles graphiquement semblables aux diagrammes chronologiques. Ces chronogrammes feuilles décrivent les événements qui sont perçus à l'interface du système sur ses canaux de communication avec l'extérieur (*les ports*). Les contraintes temporelles servent à décrire les relations entre les temps d'occurence des événements sur les ports des chronogrammes feuilles.

Une fois que sont établies les opérations de base d'un système (décrites par des chronogrammes feuilles), elles peuvent être combinées ensemble récursivement pour décrire des comportements plus complexes. Les chronogrammes hiérarchiques permettent cette composition des opérations de base. Chaque chronogramme hiérarchique est caractérisé par un opérateur (`boucle`, `enchaînement`, `choix` ou `parallélisation`) dont la sémantique décrit le genre de la composition.

Les chronogrammes feuilles composés hiérarchiquement décrivent la valeur et les relations temporelles entre les événements des ports de l'interface d'un système. Les calculs effectués par le système sont décrits en annotant de fonctions, procédures et variables d'état les chronogrammes.

Ce chapitre débute par l'introduire le concept de simulation de chronogrammes sans qu'il soit nécessaire de définir précisément les chronogrammes hiérarchiques. Le section 2 décrit les

éléments de base des chronogrammes hiérarchiques (les ports, les variables d'état, les contraintes temporelles et les annotations) et définit la sémantique de ces éléments dans le cadre de la simulation de chronogrammes. La section 3 présente ensuite comment sont formés les chronogrammes hiérarchiques annotés à partir des constituants de base.

1. La simulation de chronogrammes

Un modèle de système est conforme à sa réalisation quand le modèle répond de la même façon que la réalisation aux événements de son environnement. En assumant qu'un modèle est conforme à la réalisation du système qu'il décrit, il suffit de prouver la fonctionnalité de son modèle pour prouver la fonctionnalité d'un système. Une des manières de se convaincre que la fonctionnalité d'un modèle est exacte est de le simuler le plus exhaustivement possible.

L'acte de simuler consiste à exécuter le modèle d'un système dans des conditions telles qu'elles exercent une partie ou toute la fonctionnalité du modèle. Par exemple, simuler le modèle d'un boîtier de mémoire pourrait consister à y écrire et y lire successivement des valeurs. Pour ce faire, il faut générer sur les ports de l'interface du boîtier de mémoire les événements menant à l'exécution par le modèle de cycles d'écriture et de lecture.

Le modèle d'un système qu'il est possible d'exécuter s'appelle un *modèle exécutable*. Les modèles de système décrits par des chronogrammes hiérarchiques annotés sont des modèles exécutables. A l'aide de l'outil présenté dans le chapitre 3, il est possible de transformer un modèle décrit par un chronogramme hiérarchique annoté en un modèle exécutable dans le langage de description de matériel VHDL.

1.1. Événements spécifiés et événements réels. Il est utile pour la compréhension de l'outil de simulation de distinguer les événements de la spécification des événements ayant lieu durant la simulation.

Le premier type d'événements, les événements spécifiés, sont ceux qui composent la spécification d'un système. Le modélisateur se sert des diagrammes chronologiques pour décrire des séquences d'événements spécifiés qu'il est possible de percevoir sur les ports de l'interface d'un système.

Le deuxième type d'événements, les événements réels, sont les événements perçus sur les ports de l'interface du modèle du système lorsque celui-ci est exécuté.

1.2. Exécution d'un modèle. Pendant l'exécution d'un chronogramme hiérarchique annoté, l'environnement du modèle dépose des événements sur les ports de l'interface du modèle d'un système. Selon les séquences d'événements qui lui sont soumises, le modèle génère les

réponses à ces stimuli. Les événements réels sur les ports possèdent une valeur d'un certain domaine. Un domaine possible est l'ensemble des valeurs logiques booléennes 0 et 1.

Les fonctions que remplit le modèle exécutable sont énumérées dans les trois sous-sections suivantes.

1.2.1. *Validation de la valeur des événements réels.* Pendant la simulation, le modèle valide la valeur des événements réels générés par l'environnement. Cela consiste à comparer la valeur des événements réels à celle des événements spécifiés. Un chronogramme hiérarchique annoté décrit des séquences d'événements spécifiés qu'il s'agit de faire correspondre à des événements réels durant la simulation.

1.2.2. *Validation du temps d'occurence des événements réels.* Les contraintes temporelles des chronogrammes hiérarchiques annotés permettent de préciser les relations temporelles entre les événements spécifiés. Le modèle exécutable s'assure que ces relations sont respectées par les événements réels. Quand une relation temporelle n'est pas respectée, l'exécution du modèle est avortée et l'erreur qui est survenue est indiquée.

1.2.3. *Génération des réponses du système.* Jusqu'à présent le rôle d'un modèle exécutable n'a été que de valider la valeur et le temps d'occurence des événements réels durant la simulation. La dernière fonction du modèle exécutable sera donc de générer les réponses du système aux stimuli de son environnement. Cela consiste à la génération par le modèle d'événements réels d'une certaine valeur et à un temps précis.

La valeur de ces événements est calculée grâce aux événements spécifiés et aux annotations tandis que leur temps d'occurence est tel qu'il respecte les délais entre les événements dictés par les contraintes temporelles.

La section suivante approfondit la notion d'interface de système.

2. Les constituants des chronogrammes hiérarchiques annotés

Lorsqu'un système est décrit, il faut distinguer son comportement externe de son comportement interne. Dans le premier cas, on se situe à l'extérieur du système pour décrire quel est son fonctionnement. Cette description permet de répondre à des questions telles : "Quelles fonctions remplit le système?" et "Comment le système interagit avec son environnement?". Dans le deuxième cas, il s'agit de démontrer comment le système réalise les opérations qu'il supporte en proposant une réalisation. Une manière de décrire le comportement externe d'un système consiste à décrire les événements perçus pendant son fonctionnement sur son interface avec l'extérieur. Une description contient les éléments suivants:

2.1. Les ports. Les ports sont les canaux utilisés par un système pour communiquer avec son environnement. Trois caractéristiques définissent l'usage d'un port: sa direction, le type des données qu'il transporte et son interprétation.

L'ensemble des ports d'un système forme une interface entre le système et son environnement. L'interaction entre ces deux entités prend la forme d'échanges d'événements sur les ports. Les chronogrammes hiérarchiques annotés décrivent précisément ces échanges d'information entre le système et son environnement.

2.1.1. *Direction d'un port.* La direction d'un port indique d'où proviennent les événements qu'il transporte. Trois valeurs de direction sont possibles :

entrée: cette direction signifie que les événements que transmet le port émanent de l'environnement et sont destinés à être envoyés au système;

sortie: les événements sont cette fois produits par le système et sont destinés à l'environnement;

entrée-sortie: le port peut véhiculer des événements en entrée et en sortie. Un port n'est utilisé que dans une direction à la fois et donc ne peut pas être au même instant utilisé en entrée et en sortie.

2.1.2. *Type d'un port.* Le domaine des valeurs de l'information transmise par un port s'appelle le type du port. Comme un chronogramme va être plus tard traduit dans le langage VHDL pour créer une spécification exécutable, les types de port permis sont ceux du dit langage. Cela inclut : les types les plus courants des langages de programmation en plus des types pouvant représenter les valeurs possibles de signaux numériques et des types abstraits comme des enregistrements.

2.1.3. *Interprétation d'un port.* L'interprétation d'un port modifie la définition de ce qu'est un événement sur un port.

DÉFINITION 2.1. *Interprétation signal. Un événement a lieu sur un port signal quand la valeur transmise par celui-ci change. Ainsi, il suffit d'observer un changement de valeur du port pour conclure qu'un événement a lieu.*

DÉFINITION 2.2. *Interprétation message. L'interprétation message d'un port est un peu plus subtile. Un événement est observé sur un port message dès que la source du port y dépose une valeur. Ce type d'événement ne donne pas lieu nécessairement à un changement de valeur du port parce que la nouvelle valeur déposée peut être équivalente à l'ancienne. Cette différence*

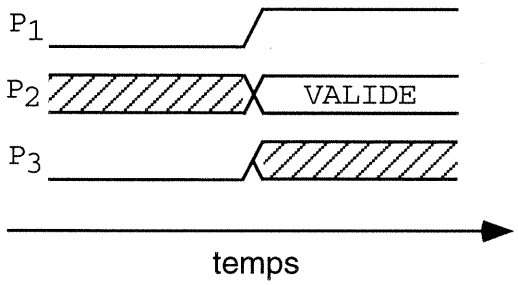


FIGURE 2.2. Représentation graphique des événements spécifiés

d'interprétation s'avère utile pour la modélisation de système à haut niveau où par exemple il est question de transmission de message.

2.1.4. *Les états d'un port.* À tout instant un port se trouve dans l'un des trois états suivants.

DÉFINITION 2.3. *L'état constant. Dans cet état, le port transmet une information constante dont la valeur est connue. La valeur de la constante est dans le domaine du type du port.*

DÉFINITION 2.4. *L'état valide. Cet état est semblable à l'état constant. La différence entre cet état et l'état constant est que dans le cas de l'état valide, la valeur transportée par le port est inconnue, au moment d'écrire la spécification.*

DÉFINITION 2.5. *L'état indéfini. Dans cet état, un port peut avoir un comportement quelconque, stable ou changeant.*

2.2. Événements spécifiés. Le passage d'un état à un autre d'une spécification d'un port constitue un événement de la spécifié. Le genre des événements peut être soit constant, soit valide ou indéfini et est déterminé par l'état que prend le port après l'événement.

DÉFINITION 2.6. *Événement constant. Le passage d'un port de n'importe quel état à l'état constant constitue un événement constant.*

DÉFINITION 2.7. *Événement valide. Le passage d'un port de n'importe quel état à l'état valide constitue un événement valide.*

DÉFINITION 2.8. *Événement indéfini. Le passage d'un port de n'importe quel état à l'état indéfini constitue un événement indéfini.*

La figure 2.2 contient la représentation graphique de différents événements spécifiés. L'axe horizontal de la figure représente le temps et l'axe vertical représente des ports dont le type est booléen. Le premier des ports (P_1) a initialement la valeur logique 0 (barre horizontale inférieure) et après un événement constant, il passe à la valeur logique 1 (barre horizontale supérieure). Le port P_2 est initialement à l'état indéfini puis passe à l'état valide, constituant ainsi un événement valide. La section de temps où il est spécifié que le port est à l'état indéfini est représentée par la région hachurée et la section valide est représentée par la section qui la suit. Le port P_3 contient un événement indéfini, car le port passe de l'état constant (valeur logique 0), à l'état indéfini (région hachurée).

2.3. Contraintes temporelles entre événements. L'ordre selon lequel sont spécifiés des événements sur un même port indique dans quel ordre ils seront observés. Ceci ne permet que de mettre en relation entre eux des événements sur un même port. De plus, cela ne permet pas de spécifier quantitativement le temps s'écoulant entre l'observation de deux événements. Pire encore, il n'est pas possible de spécifier l'ordre chronologique entre événements de ports différents. Les contraintes temporelles servent à mettre en relation temporelle des événements entre eux et ce, quels que soient les ports des événements à mettre en relation.

La notation suivante est utilisée dans la définition des contraintes temporelles. Si a et b sont des événements spécifiés alors t_a et t_b sont les temps respectifs où sont observés ces événements.

2.3.1. *Contraintes* `assume` *ou* `commit`. Il existe deux genres de contraintes temporelles: les contraintes dites `assume` et les contraintes dites `commit`. Les premières servent à mettre en relation des événements quelconques avec des événements d'entrée de l'interface. Leur rôle est de spécifier les bornes temporelles à l'intérieur desquelles ces événements doivent être observés. Quant aux contraintes `commit`, elles servent à mettre en relation des événements quelconques avec des événements de sortie et leur rôle est de spécifier les bornes temporelles à l'intérieur desquelles ces événements doivent être générés par le système.

2.3.2. *Ordre entre deux événements.* La relation temporelle (`precedence` *intent* `a b l u`) exprime la relation temporelle suivante entre les temps où sont observés ou générés les événements a et b. Le symbole *intent* prend soit la valeur `assume` ou `commit`.

$$t_b \in [t_a + l, t_a + u], (0 < l \leq u)$$

Cette relation s'exprime par deux inégalités [She88]

$$t_b - t_a \geq l$$

$$t_a - t_b \geq -u$$

DÉFINITION 2.9. *Pour une contrainte* **(precedence a b l u)**, *les événements a et b désignent respectivement* la source *et* la cible *de la contrainte.*

2.3.3. *Délais entre deux événements.* La relation temporelle **(concurrence *intent* a b -l u)** est utilisée pour exprimer que les temps d'occurence des événements a et b doivent être séparés d'au moins l et d'au plus u unités de temps.

L'inégalité suivante traduit cette relation:

$$l \leq t_b - t_a \leq u, (l \leq 0, u \geq 0)$$

2.3.4. *Les opérateurs de contraintes.* Lorsqu'un événement est la cible de plusieurs contraintes, celles-ci se combinent entre elles à l'aide des opérateurs **conjunctive**, **latest** ou **earliest** [KC93].

Dans la définition de ces opérateurs, la notation qui suit est adoptée.

- Étant donné un ensemble comportant n contraintes et dont la cible est l'action E, P_i est utilisé pour désigner une contrainte de type **precedence** et X_i est utilisé pour désigner une contrainte de type **precedence** ou **concurrence** $(i = 1, \ldots, n)$.
- E_i est la source d'une contrainte X_i.
- t_i est le temps où l'action E_i est observée.
- t est le temps où l'événement E est observé.

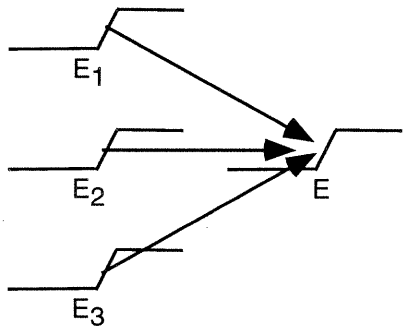


FIGURE 2.3. Composition de contraintes

DÉFINITION 2.10. *L'opérateur* **conjunctive**. *Cet opérateur permet de faire l'intersection de contraintes entre elles. Sa sémantique formelle est:*

$$\text{Conjunctive}(X_1, \ldots, Xn) : t_i + l_i \leq t \leq t_i + u_i, (i = 1, \ldots, n), (0 < l_i \leq u_i)$$

DÉFINITION 2.11. *L'opérateur* **latest**. *Cet opérateur permet d'exprimer que l'intervalle de temps dans lequel E se produit, est retardé le plus possible. Sa sémantique formelle est:*

$$\text{Latest}(P_1, \ldots, P_n) : \max_{i=1\ldots n}(t_i + l_i) \leq t \leq \max_{i=1\ldots n}(t_i + u_i), (i = 1, \ldots, n), (0 < l_i \leq u_i)$$

DÉFINITION 2.12. *L'opérateur* **earliest**. *Cet opérateur permet d'exprimer que l'intervalle de temps dans lequel E se produit est devancé le plus possible. Sa sémantique formelle est:*

$$\text{Earliest}(P_1, \ldots, P_n) : \min_{i=1\ldots n}(t_i + l_i) \leq t \leq \min_{i=1\ldots n}(t_i + u_i), (i = 1, \ldots, n), (0 < l_i \leq u_i)$$

Des définitions des opérateurs de contrainte découlent les conséquences suivantes qui restreignent les combinaisons de types de contraintes qu'il est possible de composer:

(i) il n'est pas possible de composer entre elles des contraintes **concurrence** et **precedence**;

(ii) les contraintes **concurrence** ne se composent qu'à l'aide de l'opérateur **conjunctive**.

2.3.5. *Contraintes implicites d'un chronogramme.* Pour définir ce que sont les contraintes implicites, posons que l'événement D est l'événement départ d'un chronogramme feuille, que l'événement F est l'événement fin du même chronogramme et que les événements $D_i (1 \leq i \leq n)$ sont les premiers événements de l'ensemble des ports du chronogramme dont le nombre d'éléments équivaut à n. De même les événements $F_i (1 \leq i \leq n)$ sont les derniers événements des ports du chronogramme.

(i) Il existe les contraintes (**precedence D** D_i **0**), $(1 \leq i \leq n)$. Ces contraintes sont dues au fait que l'événement départ d'un chronogramme a lieu avant les premiers événements spécifiés des ports du chronogramme.

(ii) Il existe la contrainte (**latest (precedence** F_1 **F 0 0) (precedence** F_2 **F 0 0)** ... **(precedence** F_n **F 0 0)**). Cette contrainte est due au fait que l'événement fin d'un chronogramme feuille a lieu immédiatement après le dernier des événements spécifiés du chronogramme.

(iii) De plus, si A est un événement autre que l'événement F ou D, A' est l'événement suivant A et il existe une relation (**precedence A A' 0**). Dans le cas où A' est la cible d'une autre contrainte, les deux se composent conjonctivement en une seule contrainte. Cette contrainte est due au fait que les événements spécifiés d'un même port ont nécessairement lieu les uns après les autres.

2.4. Événements observables et non-distinguables. Il est utile de scinder en deux classes les événements spécifiés d'entrée: les événements spécifiés observables et les non-distinguables. Ces deux classes sont disjointes donc tout événement spécifié de direction entrée est soit dans l'une ou l'autre de ces classes.

Les événements spécifiés observables sont ceux qui, dans le cas d'un port ayant l'interprétation signal, définissent le passage du port de l'état constant à l'état constant. Dans le cas d'un port ayant l'interprétation message, cette définition s'étend alors aux combinaisons du passage de l'état valide ou constant à l'état valide ou constant.

La définition des événements observables est, pour un port d'interprétation signal, tout événement définissant le passage d'un port à l'état valide ou indéfini à n'importe quel autre état. Dans le cas de l'interprétation message d'un port, la définition se restreint aux événements définissant le passage de l'état indéfini à l'état valide et vice-versa.

2.4.1. *Le problème des événements non-distinguables.* Lors de la simulation, le modèle exécutable fait correspondre les événements réels aux événements spécifiés. Ce propos est illustré dans le cas d'un port P dont le comportement est spécifié par la figure 2.4, composé de deux événements observables.

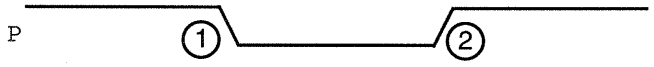


FIGURE 2.4. Spécification du port P

À la simulation, le premier événement réel correspondra nécessairement à l'événement spécifié 1 et le second à l'événement spécifié 2. Il est simple dans ce cas de faire correspondre les événements réels aux événements spécifiés.

La figure 2.5 représente la spécification d'un port signal P' comportant un seul événement appartenant à la classe des non-distinguables.

FIGURE 2.5. Spécification du port P'

À la simulation, la figure 2.6 décrit la valeur et le temps d'observation des événements sur ce port. L'axe horizontal représente le temps et l'axe vertical une spécification de ports et des comportements de ports observés durant une simulation. La spécification du port P' indique qu'il commence par avoir un comportement qui n'est pas défini puis éventuellement le port passe à l'état logique 0, à la suite d'un événement constant de valeur logique 0. Le problème est qu'il n'est pas clair à quel événement réel correspond cet événement spécifié. Dans la figure, le

comportement du port C_1 sur lequel quatre événements réels sont observés, peut correspondre au comportement spécifié du port P'. Dans ce cas, il faudrait que le modèle fasse correspondre le dernier des événements réels du port C_1 à l'événement spécifié du port P'. Le port C_2 est constamment à la valeur logique 0 et aucun événement n'y est observé. Ce comportement est conforme à la spécification du port P' mais aucun événement réel ne peut correspondre à l'événement spécifié du port P'.

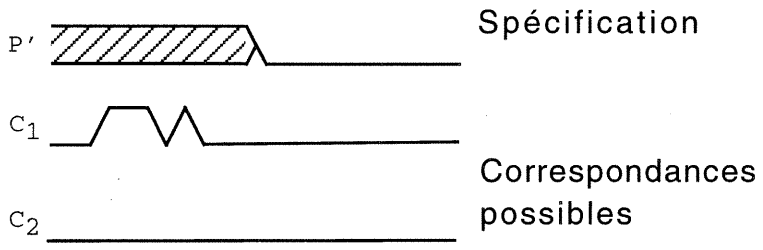


FIGURE 2.6. Correspondances possibles entre événements spécifiés et observés

L'existence dans la spécification de zones pendant lesquelles le comportement d'un signal est spécifié indéfini implique qu'il existe des intervalles de temps à la simulation pendant lesquels le port peut avoir n'importe quel comportement, ce qui est équivalent à un nombre indéterminé d'événements réels de valeur arbitraire.

Par ailleurs, les événements valides d'un port d'interprétation signal sont indistinguables parce que ces événements peuvent simplement ne pas se produire pendant la simulation et la spécification serait respectée, comme dans le cas du port P' et du comportement réel du port C_2.

Pour pallier au problème des événements non-distinguables, il faut pouvoir identifier à quel moment un signal entre et sort d'une zone de temps où son comportement est indéfini ou valide. Pour les zones valides, le modèle exécutable doit s'assurer que le comportement du port est stable (qu'il n'a subi aucune transition). Pour les zones indéfinies, le modèle permet toutes les transitions possibles sur ce port. Il est impératif de disposer dans les deux cas de bornes inférieures et supérieures sur l'intervalle de temps de la zone valide ou indéfinie. Pour disposer de ces bornes, les règles d'édition de chronogrammes feuilles suivantes doivent être suivies (figure 2.7). Tout événement indistinguable est soit:

(i) la cible d'une contrainte de préséance dont la valeur de la borne supérieure est bornée
ou

(ii) la source d'une contrainte de préséance dont la borne inférieure est bornée.

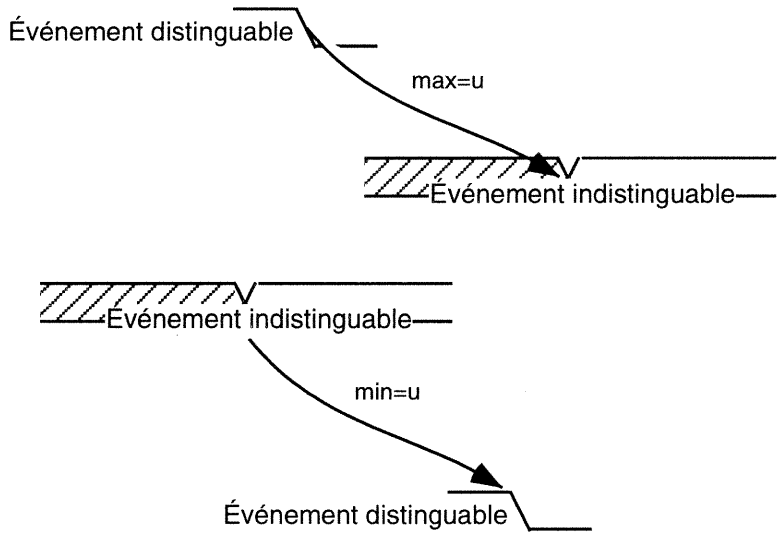


FIGURE 2.7. Règles d'édition des événements valides et indéfinis

3. LES CHRONOGRAMMES HIÉRARCHIQUES ANNOTÉS

L'interface d'un système est constituée d'un ensemble de ports servant à communiquer avec l'environnement. Décrire l'interaction entre le système et l'environnement consiste à décrire les événements entre ces deux entités observés sur les ports de l'interface du système. Les chronogrammes hiérarchiques vont servir précisément à décrire les événements à l'interface du système ainsi que les délais entre ces événements.

La spécification s'obtient en décrivant les opérations possibles à l'interface à l'aide de chronogrammes feuilles puis, ceux-ci sont combinés selon des opérateurs hiérarchiques pour former des chronogrammes hiérarchiques.

Tout chronogramme hiérarchique possède les caractéristiques suivantes:

(i) un événement de départ et un événement de fin,
(ii) un ensemble de ports externes,
(iii) un ensemble de signaux internes,
(iv) un opérateur hiérarchique.

L'événement de départ et l'événement de fin sont des événements qui délimitent le début et la fin d'un chronogramme. Dans le contexte de simulation, lorsqu'un chronogramme est démarré, l'événement de départ a lieu et similairement lorsqu'un chronogramme est terminé l'événement de fin a lieu. Ces deux types d'événements ne sont pas rattachés à un port comme les autres actions.

Les ports externes d'un chronogramme sont les ports qui constituent son interface avec le monde extérieur et ils sont visibles à l'intérieur du contexte qui instancie le chronogramme. Le contexte en question peut être un autre chronogramme ou le mécanisme de génération de modèle et dans ce cas, les ports externes deviennent les ports de l'interface du modèle généré.

Dans un langage de programmation moderne, les notions de signaux internes et de ports externes sont aux chronogrammes ce que les variables locales et les paramètres formels sont à une procédure. Un chronogramme à l'intérieur duquel un signal interne est déclaré peut seul lire et écrire une valeur sur ce signal.

3.1. Hiérarchies de chronogrammes.

3.1.1. *Les chronogrammes feuilles.* Les chronogrammes feuilles servent à décrire les opérations élémentaires à l'interface du système. La définition de ce qu'est une opération dépend de la manière dont le modélisateur conçoit le fonctionnement du système. Pour chaque opération du système, il s'agit de décrire les événements menant à l'exécution de celle-ci. Les relations temporelles entres les événements sont précisées grâce aux contraintes temporelles.

3.1.2. *Les opérateurs de composition des chronogrammes.* Les chronogrammes se composent entre eux afin de modéliser des comportements complexes de systèmes. Il existe quatre opérateurs de composition de base. Dans la définition des opérateurs hiérarchiques, la notation suivante sera utilisée. Si Q est un chronogramme alors $A_1, A_2, \ldots, A_n$ sont les n chronogrammes fils de Q.

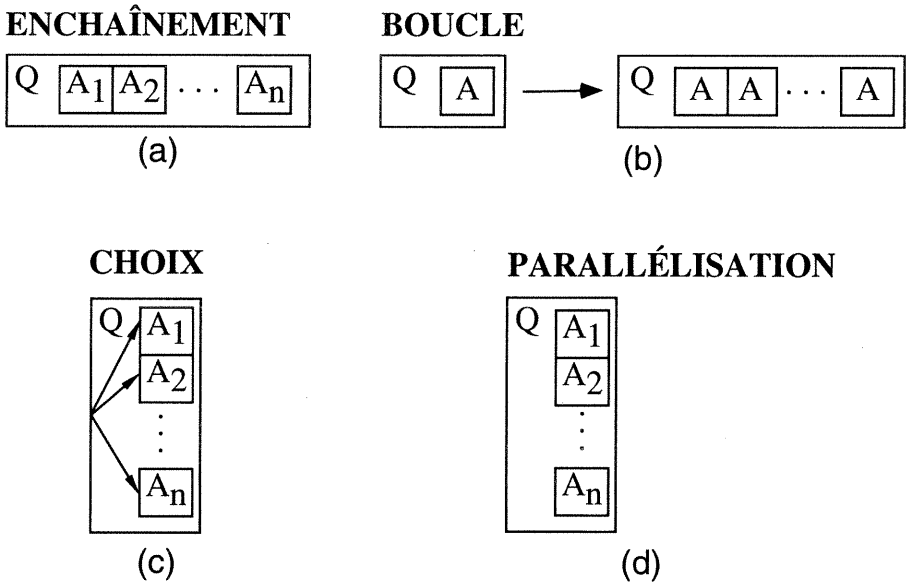


FIGURE 2.8. Composition hiérarchique de chronogrammes.

3.1.3. *L'opérateur ENCHAÎNEMENT.* La séquence d'événements que définit un chronogramme Q dont l'opérateur est l'enchaînement, est la séquence d'événements des ports de $A_1, \ldots, A_n$ dans l'ordre. L'événement de départ de A_1 coïncide avec l'événement de départ de Q tandis que l'événement de départ de A_{i+1} coïncide avec l'événement fin de A_i $(i = 1, \ldots, n-1)$. L'événement de fin de Q, quand à lui, coïncide avec celui de A_n.

3.1.4. *L'opérateur BOUCLE.* L'opérateur BOUCLE permet de boucler autour d'un chronogramme. Un prédicat de terminaison optionnel P peut être associé à Q. Ce prédicat, s'il est présent, est évalué avant chaque itération autour du chronogramme A. Lorsqu'il s'évalue à faux alors l'événement fin de Q à lieu. Si le prédicat P n'est pas spécifié alors cela équivaut à spécifier un prédicat s'évaluant toujours vrai et la séquence d'événements définie par Q est la séquence d'actions de A répétée une infinité de fois et l'événement fin de Q n'a jamais lieu.

3.1.5. *L'opérateur CHOIX.* Cet opérateur permet de choisir une séquence d'événements parmi celles définies par $A_1, \ldots, A_n$. À chaque branche A_i $(i = 1, \ldots, n)$ peut être associé un prédicat optionnel de sélection P_i et s'il est présent, alors il est évalué après l'événement de départ de Q. Les événements de départ des branches pour lesquelles le prédicat s'est évalué vrai (cela inclut les branches n'ayant pas de prédicat associé) sont exécutés. La première branche pour laquelle l'événement de fin survient est la branche définissant la séquence d'actions de Q. Dès que l'événement de fin d'une branche survient, toutes les autres branches sont abandonnées et l'événement de fin de Q est exécuté.

Si une erreur survient dans l'une des branches alors la branche en question est abandonnées. Si toutes les branches sont abandonnée alors une condition d'erreur est relevée et l'exécution du chronogrammes est avortée.

3.1.6. *L'opérateur PARALLÉLISATION.* Le comportement défini par cet opérateur est la séquence d'événements union de $A_1, \ldots, A_n$. Les événements de départ de $A_1, \ldots, A_n$ coïncident avec celui de Q. Les branches $A_1, \ldots, A_n$ sont exécutées en parallèle. L'événement fin de Q a lieu dès que tous les événements fin de $A_1, \ldots, A_n$ ont eu lieu.

3.2. L'annotation de chronogrammes. Jusqu'à maintenant, les chronogrammes ne permettent que de décrire les événements de l'interface. Il est aussi nécessaire de décrire les calculs réalisés par le système. Ceci consiste à annoter un chronogramme. Les différents types d'annotations sont : la déclaration de variables d'état, les attachements de fonctions booléennes (prédicats), les attachements de procédures et les attachements de variables automatiques.

3.2.1. *Les variables d'état d'un chronogramme.* Ce sont des variables au même sens que des variables dans un langage de programmation et elles ont le même usage que dans un programme classique. Une variable n'est visible qu'à partir du chronogramme où elle est déclarée. Les variables sont passées par référence d'un chronogramme à l'autre.

Tout comme les ports d'un modèle, les variables d'état possèdent un type servant à définir le domaine des valeurs possibles prises par les variables.

3.2.2. *Variables automatiques.* Il existe un mécanisme utile pour retenir la valeur d'un port lors de l'observation d'un événement valide d'entrée. Il suffit d'associer une variable à ce type d'événement pour qu'automatiquement la valeur du port soit retenue dans le contenu de la variable lorsque l'événement est observé. Une variable attachée à un événement valide s'appelle une variable automatique.

3.2.3. *Les attachements de fonctions.* À tous les événements d'un chronogramme (y compris l'événement de départ et de fin), il est possible d'attacher un prédicat (un appel à une fonction retournant une valeur booléenne). L'appel à ce prédicat est fait lorsque l'on tente de valider l'événement. Lorsque le prédicat s'avère faux, l'événement n'est pas validé et l'exécution du chronogramme est avortée. Par contre, lorsque le prédicat est vrai, la validation de l'événement peut se poursuivre.

3.2.4. *Les attachements de procédures.* Ici encore, à tous les événements d'une spécification, il est possible d'attacher un appel de procédure. L'appel à la procédure se fait une fois la validation de l'événement.

Les procédures d'un modèle sont utilisées pour modifier ou calculer la valeur des variables d'état du système. Par exemple, dans la figure 2.9, la procédure `read_column` est attachée à l'événement définissant la transition de 1 à 0 du signal CAS. Cette procédure modifie la variable d'état associée au paramètre formel `column` de la procédure `read_column`. Par la suite, ce résultat sera utilisé lorsque l'action valide du signal Q sera validée et que la procédure `compute_data` sera appelée.

Les fonctions et les procédures sont écrites en langage VHDL. Toutes les instructions du langage sont permises sauf l'instruction `wait` qui suspend momentanément l'exécution d'une fonction ou d'une procédure.

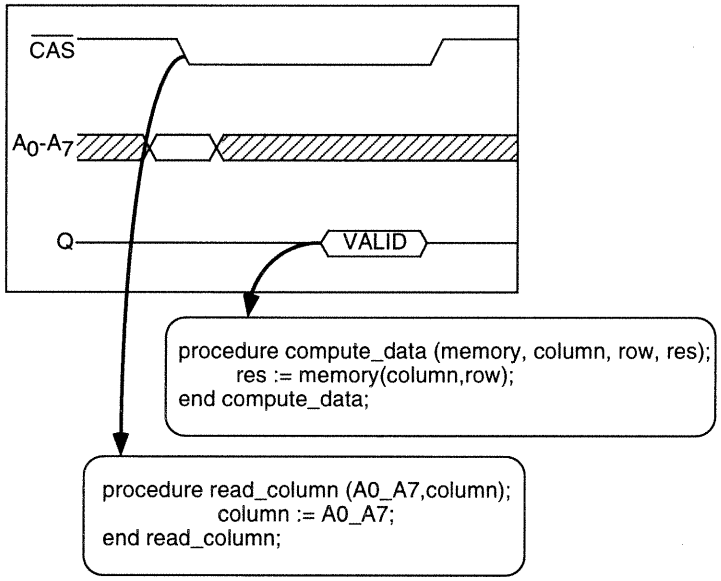


FIGURE 2.9. Exemple d'attachement de procédure

CHAPITRE 3

L'outil de spécification et de simulation

Les chronogrammes hiérarchiques permettent de spécifier des systèmes numériques. L'un des usages de ces spécifications est la vérification de systèmes. Ce chapitre présente les sous-systèmes développés pour réaliser la simulation d'un système à partir de sa spécification écrite dans le langage des chronogrammes hiérarchiques annotés. Le langage VHDL n'est pas un des sous-systèmes de l'outil en tant que tel mais comme le sous-système de simulation est écrit dans ce langage, en voici une description.

1. Le langage de modélisation VHDL

VHDL est un langage utilisé pour la modélisation de matériel numérique. L'acronyme VHDL signifie VHSIC Hardware Description Langage, dans lequel VHSIC est l'initiale de Very High Speed Integrated Circuit. La citation [ABOR90] décrit substantiellement le langage.

L'un des usages des modèles écrits à l'aide de VHDL est la simulation de systèmes. Les modèles écrits en VHDL peuvent cependant être utilisés à d'autres fins comme par exemple servir d'entrée à un système de synthèse de matériel ou à un système de vérification formelle.

La norme IEEE-1064 [gro87] définit la syntaxe et la sémantique du langage. VHDL tend à s'imposer comme standard en tant que langage de description de matériel dans l'industrie de la micro-électronique. Les principaux avantages de VHDL :

- il est un langage moderne dérivé du langage ADA; VHDL bénéficie de l'expérience acquise en génie logiciel;
- un standard rigoureux décrit la syntaxe du langage;
- commercialement, VHDL est en voie de devenir LE langage de description de matériel.

2. Définition des sous-systèmes

Au total quatre sous-systèmes composent l'outil :

(i) l'éditeur graphique de chronogrammes feuilles *AURORA*,

(ii) l'*interpréteur* du langage des chronogrammes hiérarchiques annotés,

(iii) le générateur de code,

(iv) la librairie d'exécution VHDL.

Les deux premiers sont des sous-systèmes généraux parce qu'ils ne sont pas spécifiques à l'application de simulation des chronogrammes. Ils seront partagés avec d'autres outils de vérification découlant de travaux connexes à ceux présentés dans ce mémoire. Les deux derniers sous-systèmes sont spécifiques à l'application de simulation.

La figure 3.10 illustre les sous-systèmes et l'interaction entre ceux-ci. Les sections qui suivent décrivent les sous-systèmes de l'outil.

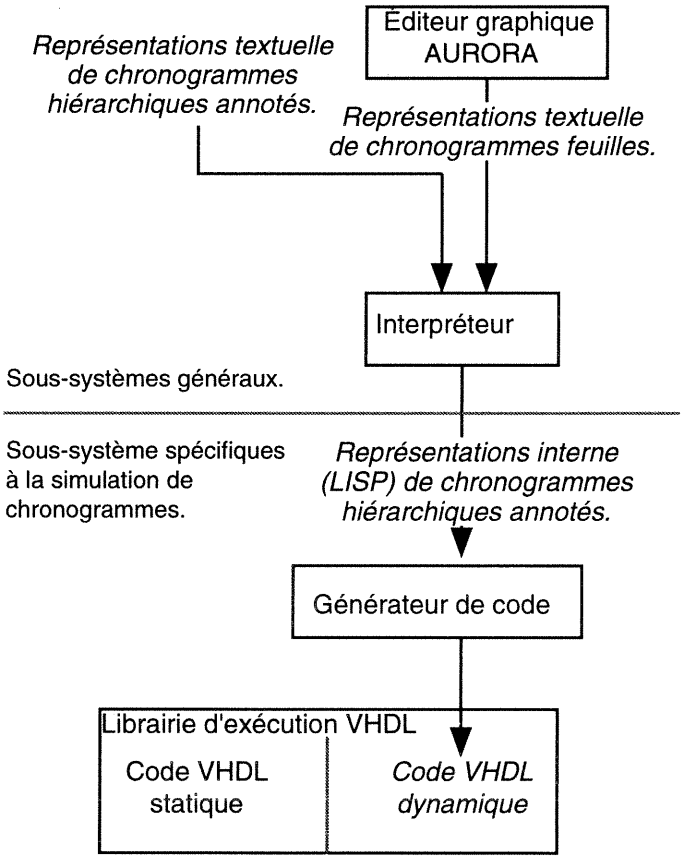


FIGURE 3.10. Les sous-systèmes de l'outil.

2.1. Outil de capture graphique de chronogrammes feuilles. Les chronogrammes sont écrits dans un langage décrit par la grammaire des chronogrammes. Les descriptions données dans ce formalisme servent de point d'entrée au reste des sous-systèmes de l'outil. Dans

le contexte d'une utilisation industrielle, ce formalisme s'avère lourd et complexe. Pour faciliter l'écriture et l'édition de chronogrammes, il est utile de disposer d'interfaces graphiques conviviales.

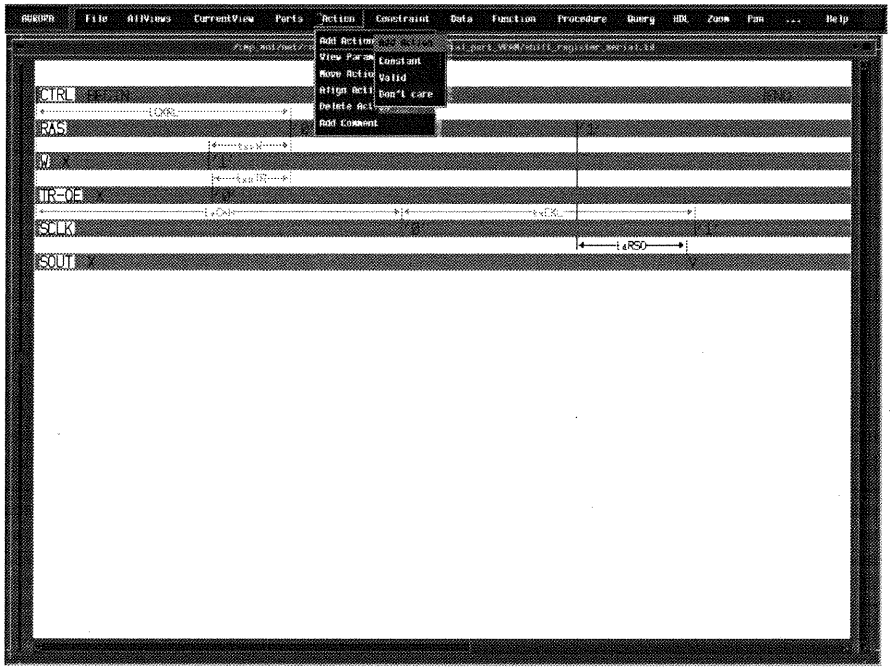


FIGURE 3.11. L'éditeur de chronogrammes feuilles AURORA.

Le premier système qui a été développé dans ce sens est AURORA. Il s'agit d'un éditeur graphique de chronogrammes feuilles. AURORA est un logiciel dérivé de l'éditeur de chronogrammes SHADOW, propriété de la compagnie *Recherches Bell-Northern Ltée.*

La figure 3.11 est tirée d'une session de travail avec l'outil AURORA. La fenêtre d'édition de chronogrammes contient les éléments suivants: les ports du chronogramme sont disposés verticalement. Sur les ports sont positionnés horizontalement les événements. Les contraintes temporelles sont représentées par les flèches entre les événements. Finalement, les attachements de procédures/fonctions sont illustrés par le texte adjacent aux événements. Le texte représente l'appel fait à la procédure ou à la fonction qui lui est adjacente.

AURORA permet de manipuler interactivement les éléments précédents de manière à éditer des chronogrammes hiérarchiques. L'interface usager utilise la souris et des menus déroulants de manière à rendre conviviale l'interaction avec l'usager. Le résultat d'une session d'édition peut être sauvegardé pour être réédité ultérieurement. Finalement, une fonction d'AURORA

permet d'exporter un chronogramme (feuille) dans une forme textuelle pour qu'il puisse être incorporé à des descriptions hiérarchiques.

Les descriptions de chronogrammes feuilles exportées d'AURORA et les chronogrammes hiérarchiques sont quant à eux capturés sous forme textuelle par l'utilisateur. Ces descriptions de chronogrammes servent d'entrée au sous-système suivant.

2.2. Interpréteur du langage des chronogrammes hiérarchiques annotés. La fonction principale de ce sous-système est de construire une représentation interne des chronogrammes qui lui sont présentés. Cette représentation interne est ensuite utilisée par le sous-système générateur de code.

Le langage d'entrée de l'interpréteur est le langage des chronogrammes hiérarchiques annotés dont la syntaxe est donnée en annexe A de ce mémoire. Ce langage permet d'exprimer tous les types de chronogrammes hiérarchiques (feuille, enchaînement, boucle, choix et parallélisation) de même que les différents types d'annotations (attachements de fonctions booléennes, attachements de procédures, attachements de variables d'état et attachements de variable automatiques).

En plus de fournir un moyen d'exprimer les chronogrammes, l'interpréteur possède un mécanisme de définition de chronogrammes. Sa raison d'être est la même que celle des mécanismes de définition de fonctions ou de procédures dans les langages de programmation. La forme du langage qui permet les définitions s'appelle la forme *DEFBEHAVIOR*. Son utilisation précise est démontrée dans le chapitre 4.

L'interpréteur et le générateur de code sont tous les deux réalisés dans le même langage (LISP). Le passage des données de l'interpréteur au générateur se fait d'une façon transparente pour l'utilisateur en mémoire de l'ordinateur.

2.3. Générateur de code. Le générateur de code est un sous-système qui accepte en entrée une représentation d'un chronogramme hiérarchique annoté et produit en sortie une partie du code VHDL nécessaire pour simuler le comportement qu'il décrit. Les données en entrée du générateur de code proviennent de la sortie de l'*interpréteur*. La sortie du générateur consiste en du code en langage VHDL et un script en langage de la coquille C du système d'exploitation UNIX. Le code VHDL est destiné à être compilé et relié à une librairie VHDL pour constituer le modèle exécutable du chronogramme hiérarchique initial. Le script généré contient les directives de compilation nécessaires pour compiler le code généré.

2.4. Librairie d'exécution en VHDL. La librairie d'exécution en VHDL est le dernier des sous-systèmes de l'outil. Il s'agit de plusieurs unités VHDL qui sont utilisées pour la

simulation des chronogrammes. Une partie de cette librairie est statique donc elle demeure la même peu importe le modèle simulé. L'autre partie de la librairie est qualifiée de dynamique parce qu'elle diffère d'un modèle à l'autre. La partie dynamique de la librairie est générée par le sous-système générateur de code. Ensemble, la partie statique et la partie dynamique composent la description nécessaire au simulateur du langage VHDL.

Ce chapitre a décrit les sous-systèmes de l'outil de spécification et de simulation ainsi que leur interaction. L'usage de l'outil va maintenant être illustré à l'aide d'un exemple dans le prochain chapitre.

CHAPITRE 4

Exemple d'application de la méthodologie

Ce chapitre illustre le processus menant à l'obtention d'un modèle exécutable à partir de la spécification d'un système.

Le système modélisé est une file d'attente et est inspiré d'un système de traitement de messages dans le domaine de la télécommunication. Les opérations de base du modèle sont capturées grâce à l'éditeur de chronogrammes feuilles AURORA. La composition hiérarchique de ces opérations est, quant à elle, capturée directement dans le langage de description des chronogrammes. Une fois les étapes de capture réalisées, le modèle exécutable en VHDL est produit et simulé.

1. Spécification du système modélisé

Le système proposé dans cet exemple n'est pas particulièrement complexe, néanmoins, les principaux concepts de la méthodologie de modélisation y sont inclus. Le système est une file d'attente dont les éléments arrivent à un rythme irrégulier et en sortent à un rythme régulier. Les éléments traités par la file sont des messages et il en existe deux types: les vides et les non-vides. Le système possède deux ports : l'un qui accepte les messages dans la file (`INBOX`), l'autre qui extrait les messages de la file (`OUTBOX`).

Les messages sont admis à un taux maximal d'un à tous les quinze nanosecondes et un taux minimal d'un message à tous les cinquante nanosecondes. Seuls les messages non-vides sont admis dans la file. Les messages sortent de la file à un taux régulier et lorsque vient le moment d'extraire un message de la queue et que celle-ci est vide alors un message vide est produit et déposé sur le port de sortie `OUTBOX`. Il est supposé que la queue peut contenir un nombre maximal de messages fixé arbitrairement à 200.

2. Modèle d'une file

La file réalise deux opérations fondamentales à son interface : la lecture de messages sur son port `INBOX` et l'écriture de messages sur son port `OUTBOX`. Le modèle de la queue est divisé en deux parties concurrentes : l'une qui modélise les opérations de lecture et l'autre les opérations d'écriture.

La fonctionnalité de ce modèle est réalisée en annotant les chronogrammes des éléments suivants :

(i) La procédure `INIT_QUEUE` servant à initialiser à la valeur d'une queue vide, la variable `QUEUE`.

(ii) La procédure `ENQUEUE` dont les paramètres formels sont une variable `Q` de type `QUEUE_TYPE` et un signal `S` de type `CELL_TYPE`. Cette procédure extrait de la tête de la queue un message et l'écrit sur le port `Q`.

(iii) La procédure `DEQUEUE` dont les paramètres formels sont les même que pour la procédure `ENQUEUE`. Cette procédure insère à la fin de la queue une cellule dont la valeur est lue sur le port `Q`.

Les unités de conception VHDL définissant les éléments précédents ainsi que les types des signaux et des variables utilisés dans le modèle de la file sont présentées dans la figure 4.12 et 4.13. Il incombe à l'utilisateur de définir et d'écrire ces définitions de type et les procédures utilisées pour la modélisation de la queue.

2.1. La lecture de messages. L'opération de lecture d'un message consiste à lire la valeur d'un message sur le port `INBOX` lorsque celui-ci y est déposé par l'environnement. Ensuite, si le message n'est pas vide, le système l'insère à la fin de la file d'attente.

Le comportement de lecture du modèle de la file consiste en une séquence infinie d'opérations de lecture de messages. Il existe une relation de préséance *ASSUME* entre deux lectures de messages dont les bornes *min* et *max* sont dérivées du taux d'arrivée minimal et maximal des messages. Le relation sert à vérifier que les messages sont admis dans la file à la cadence permise.

L'opération de lecture d'un message est décrite par le chronogramme feuille `READ_A_CELL`. La séquence infinie de lecture de messages que le système effectue s'obtient en composant hiérarchiquement une boucle infinie autour de l'opération de lecture.

Le chronogramme feuille `READ_A_CELL`, tel que capturé avec AURORA, est illustré dans la figure 4.14.

```
package RAQ_TYPE is
  – Definition des genres de message
  type GENRE_MESSAGE is (VIDE, NON_VIDE);
  – Definition du type message
  type MESSAGE_TYPE is record
    ID : GENRE_MESSAGE; – Ce champs definit le genre du message
    CHARGE : integer; – Ce champs definit le contenu du message
  end record;
  – Definition d'un message vide
  constant MESSAGE_VIDE : MESSAGE_TYPE := (VIDE, 0);

  constant LONGUEUR_QUEUE : integer := 200;
  subtype MESSAGE_RANGE is integer range 0 to LONGUEUR_QUEUE - 1;
  type MESSAGE_ARRAY is array (MESSAGE_RANGE) of MESSAGE_TYPE;

  – Definition du type queue
  type QUEUE_TYPE is record
    DEBUT : MESSAGE_RANGE;
    FIN : MESSAGE_RANGE;
    FIFO : MESSAGE_ARRAY;
  end record;
end RAQ_TYPE;
```

FIGURE 4.12. Package VHDL définissant les types utilisés dans le modèle de la queue.

La seule annotation présente sur ce chronogramme est l'attachement de l'appel de procédure **ENQUEUE** à l'événement associé au dépôt d'un message sur le port **INBOX**. Cette procédure accepte en entrée la valeur déposée sur le port **INBOX** et en sortie modifie la variable d'état **QUEUE**.

L'événement fin de ce chronogramme a lieu immédiatement après l'événement de dépôt d'un message. Sans perdre un instant, l'opérateur **boucle** déclenche l'événement de départ du même chronogramme mais cette fois pour l'itération suivante. La relation temporelle de préséance entre l'événement de départ du chronogramme et l'événement de dépôt d'un message établit donc la contrainte temporelle entre deux arrivées de messages successives sur le port **INBOX**.

L'exportation du chronogramme **READ_A_CELL** capturé graphiquement en une forme textuelle appropriée pour le générateur de code VHDL se fait en invoquant la commande **Save in VHDL generator format** du menu **HDL** d'AURORA. Cette commande produit le fichier **READ_A_CELL.l** dans lequel est défini le chronogramme.

2.1.1. *Le chronogramme feuille* ***READ_A_CELL.*** La syntaxe du contenu du fichier **READ_A_CELL** suit celle du langage LISP à laquelle ont été ajoutées les formes nécessaires à l'expression des objets des chronogrammes hiérarchiques annotés. La syntaxe de ces formes en BNF se trouve en annexe. Le chronogramme **READ_A_CELL** tel qu'exporté d'AURORA se trouve dans la figure 4.15.

```
library STD;
use STD.TEXTIO.all;

package body RAQ_PROCEDURE is

  -- Procedure de mise a vide d'une queue
  procedure INIT_QUEUE (QUEUE : inout QUEUE_TYPE) is
    variable LIGNE: LINE;
  begin
    QUEUE.DEBUT := 1;
    QUEUE.FIN := 0;
    write (LIGNE, now);
    write (LIGNE, string'(" Initialisation de la queue a vide."));
    writeline (OUTPUT, LIGNE);
  end INIT_QUEUE;

  -- Procedure d'ajout d'un element dans la queue
  procedure ENQUEUE (signal SIG : in MESSAGE_TYPE; variable QUEUE : inout QUEUE_TYPE) is
    variable LIGNE: LINE;
  begin
    ifSIG.ID = NON_VIDE then
     ifQUEUE.DEBUT = QUEUE.FIN then -- Y a-t-il de la place dans la queue?
      write (LIGNE, now);
      write (LIGNE, string'(" ERREUR: Debordement de la queue."));
      writeline (OUTPUT, LIGNE);
     else
      QUEUE.FIFO(QUEUE.DEBUT) := SIG;
      QUEUE.DEBUT := (QUEUE.DEBUT + 1) mod LONGUEUR_QUEUE;
      write (LIGNE, now);
      write (LIGNE, string'(" Insertion dans la queue d'un message de valeur => "));
      write (LIGNE, SIG.CHARGE);
      writeline (OUTPUT, LIGNE);
     end if;
    end if;
  end ENQUEUE;

  -- Procedure de retrait d'un element de la queue
  procedure DEQUEUE (signal SIG : out MESSAGE_TYPE; variable QUEUE : inout QUEUE_TYPE) is
    variable LIGNE: LINE;
  begin
    if (QUEUE.FIN + 1) mod LONGUEUR_QUEUE = QUEUE.DEBUT then -- Est-ce que la queue est vide?
     SIG ¡= MESSAGE_VIDE;
    else
     SIG ¡= QUEUE.FIFO(QUEUE.FIN);
     QUEUE.FIN := (QUEUE.FIN + 1) mod LONGUEUR_QUEUE;
     write (LIGNE, now);
     write (LIGNE, string'(" Extraction de la queue d'un message de valeur => "));
     write (LIGNE, QUEUE.FIFO(QUEUE.FIN).CHARGE);
     writeline (OUTPUT, LIGNE);
    end if;
  end DEQUEUE;
end RAQ_PROCEDURE;
```

FIGURE 4.13. Package VHDL définissant les procédures utilisés dans le modèle de la queue.

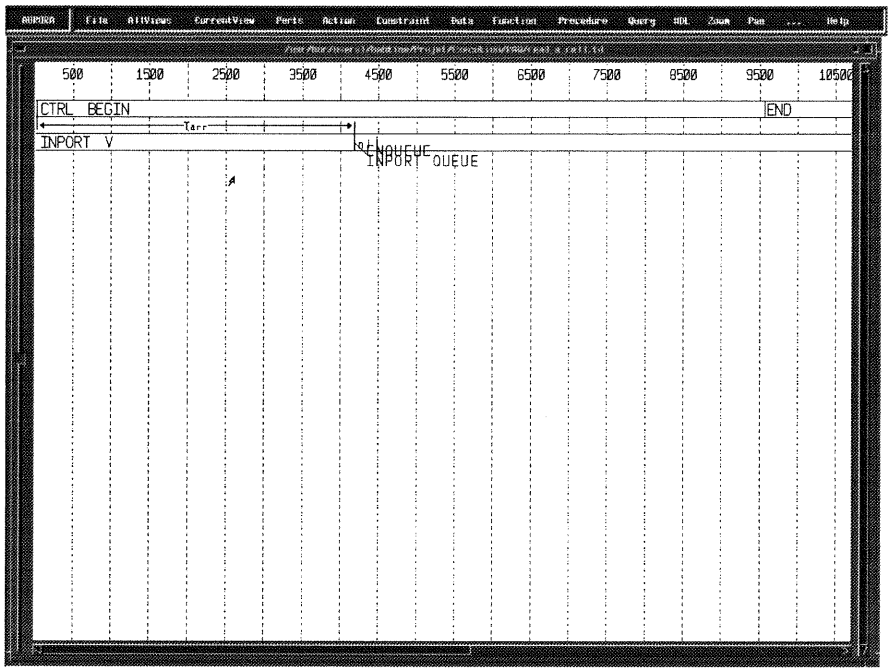


FIGURE 4.14. Chronogramme feuille décrivant l'opération de lecture

```
;;; Leaf created with AURORA
(DEFBEHAVIOR read_a_cell
 (PORTS
  (PORT INPORT IN "MESSAGE_TYPE" MESSAGE))
 (PARAMS
  (PARAM QUEUE INOUT "QUEUE_TYPE"))
 (LEAF
  (CARRIER-SPEC INPORT
                (INITIAL-SPEC (VALID))
                (ACTION-SPEC 'ev9626592 (VALID) (DIRECTION IN)
                 (PROCEDURE-CALL "ENQUEUE" INPORT QUEUE)))
  (PRECEDENCE 'start_a 'ev9626592 (CMIN 15) (CMAX 50) (INTENT ASSUME))
))
```

FIGURE 4.15. Le chronogramme **READ_A_CELL** tel qu'exporté d'AURORA

Dans la figure 4.15, la forme DEFBEHAVIOR indique qu'il s'agit d'une définition de comportement dans le langage des chronogrammes. Les comportements deviennent des chronogrammes lorsqu'ils sont *instanciés* ou, plus précisément, lorsqu'ils sont utilisés dans la définition d'autres comportements et lorsqu'ils sont transformés en code VHDL. Les formes PORTS et PARAMS servent à déclarer l'interface du comportement. Dans ce cas-ci, l'interface consiste en un port INPORT et une variable d'état QUEUE. La forme PORT indique qu'il s'agit du port formel INPORT de directionnalité IN, de type MESSAGE_TYPE et d'interprétation MESSAGE. Similairement, la forme PARAM indique que l'unique paramètre de ce comportement se nomme QUEUE, qu'il est de directionnalité INOUT et de type QUEUE_TYPE.

La dernière forme comprise dans la forme DEFBEHAVIOR correspond au corps de la définition du comportement. La forme LEAF sert à définir un chronogramme feuille. A son tour, cette forme contient les définitions des signaux et des contraintes du comportement.

Le comportement du signal INPORT est décrit par la forme CARRIER-SPEC. Il y est indiqué que l'état initial du signal est valide et qu'un seul événement spécifié se retrouve sur ce signal. La forme ACTION-SPEC vient quant à elle spécifier les attributs de cet événement. AURORA a donné à cet événement le nom ev9626592. Il est ensuite indiqué que c'est un événement d'entrée valide. Finalement, l'appel à la procédure ENQUEUE est spécifié par la forme PROCEDURE-CALL.

La seconde partie de la forme LEAF est la description de la contrainte temporelle entre l'événement de départ et l'événement sur le port INPORT.

2.2. L'écriture de messages. L'opération d'écriture de messages sur le port de sortie est similaire à l'opération de lecture d'un message. Le chronogramme feuille WRITE_A_CELL décrit l'opération d'extraction d'un message de la queue et son écriture sur le port OUTBOX. La séparation temporelle entre deux écritures de message est représentée par la contrainte temporelle de type COMMIT entre l'événement de départ du chronogramme et l'opération d'écriture. Cette contrainte est de type COMMIT parce qu'elle a pour cible un événement de sortie du système.

Le comportement WRITE_A_CELL, tel que capturé à l'aide de l'outil AURORA, est illustré dans la figure 4.16. La seule annotation contenue dans cette feuille consiste en l'attachement de la procédure DEQUEUE à l'événement d'écriture d'un message.

La définition de ce chronogramme telle qu'obtenue par AURORA suit dans la figure 4.17. Cette description est similaire à celle du chronogramme READ_A_CELL.

Comme l'opération d'écriture se produit indéfiniement, elle sera englobée par l'opérateur hiérarchique boucle (haad-loop), similairement à l'opération de lecture d'un message.

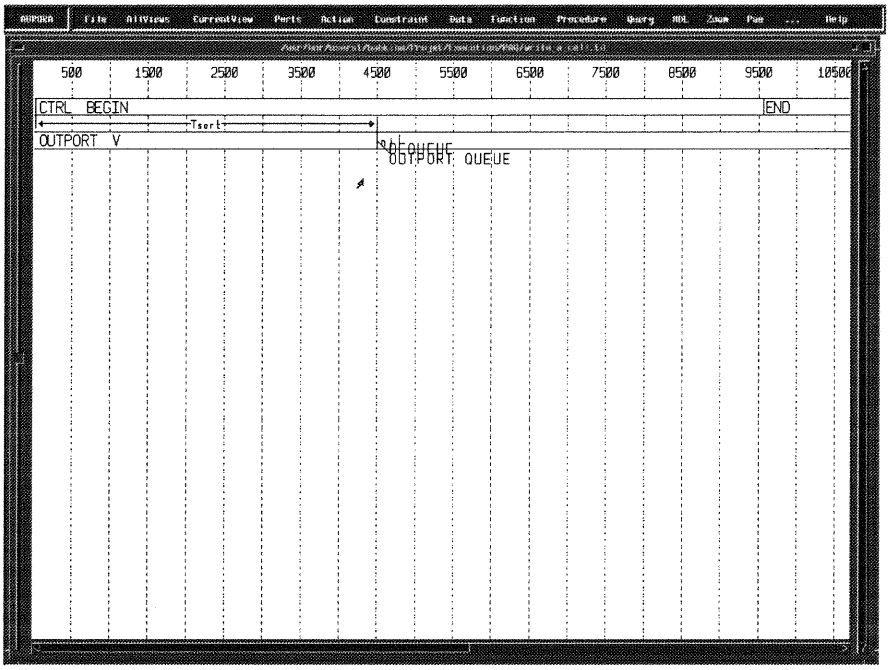


FIGURE 4.16. Chronogramme feuille décrivant l'opération d'écriture

```
;;; Leaf created with AURORA
(DEFBEHAVIOR write_a_cell
   ( PORTS
    (PORT OUTPORT OUT "MESSAGE_TYPE" MESSAGE))
   (PARAMS
    (PARAM QUEUE INOUT "QUEUE_TYPE"))
   (LEAF
    (CARRIER-SPEC OUTPORT
                  (INITIAL-SPEC (VALID))
                  (ACTION-SPEC 'ev9794528 (VALID) (DIRECTION OUT)
                   (PROCEDURE-CALL "DEQUEUE" OUTPORT QUEUE)))
    (PRECEDENCE 'start_a 'ev9794528 (CMIN 20) (CMAX 20) (INTENT COMMIT))
))
```

FIGURE 4.17. Le comportement **WRITE_A_CELL**

```
;; Definition du chronogramme Rate Adaptation Queue (RAQ)
(DEFBEHAVIOR RAQ
  (PORTS (PORT INPORT IN "MESSAGE_TYPE" MESSAGE)
         (PORT OUTPORT OUT "MESSAGE_TYPE" MESSAGE))
  (VAR QUEUE "QUEUE_TYPE")
  (PARALLEL
   (HAAD-LOOP
    (BEHAVIOR the_read read_a_cell (PORT-MAP INPORT) (PARAM-MAP QUEUE)))
   (HAAD-LOOP
    (BEHAVIOR the_write write_a_cell (PORT-MAP OUTPORT) (PARAM-MAP QUEUE)))
```

FIGURE 4.18. Définition du comportement **RAQ**

2.3. Chronogramme hiérarchique de la queue. Les deux sections précédentes ont décrit les comportements respectifs des ports **INBOX** et **OUTBOX** du système. Il reste maintenant à joindre ces comportements pour former la description complète du système.

Les ports **INBOX** et **OUTBOX** ont un comportement indépendant l'un de l'autre lorsque considérés du point de vu de l'interface du système. Le comportement de l'un se déroule parallèlement au comportement de l'autre donc leur comportement respectif doit donc être mis en parallèle. Ceci se fait en créant un nouveau chronogramme hiérarchique englobant des boucles infinies autour d'instances des comportements **WRITE_A_CELL** et **READ_A_CELL**. L'opérateur hiérarchique de ce nouveau chronogramme est l'opérateur **PARALLEL**. La définition de ce comportement telle qu'écrite par l'utilisateur suit dans la figure 4.18.

L'interface du comportement **RAQ** contient cette fois 2 ports: **INPORT** et **OUTPORT**. La forme **VAR** sert à la déclaration de la variable d'état **QUEUE** de type **QUEUE_TYPE**. Le corps de la définition de ce comportement consiste en la mise en parallèle de deux boucles autour des chronogrammes feuilles **READ_A_CELL** et **WRITE_A_CELL** respectivement.

La forme **BEHAVIOR** sert à l'instanciation d'un chronogramme déjà défini par la forme **DEFBEHAVIOR**. Dans ce cas ci, deux instanciations sont réalisées: la première est l'instanciation du comportement **read_a_cell** et la seconde est l'instanciation du comportement **write_a_cell**. Les deux instanciations engendrent chacune un chronogramme dont les noms respectifs sont **the_read** et **the_write**.

L'intérieur des formes **BEHAVIOR** utilise les formes **PORT-MAP** et **PARAM-MAP**. Elles servent à associer les éléments de l'interface formelle des comportements définis par **READ_A_CELL** et **WRITE_A_CELL** aux éléments effectifs du comportement **RAQ**.

La définition du chronogramme **RAQ** est composée de la mise en parallèle des comportements des ports **INBOX** et **OUTBOX**. Ceux-ci sont les répétitions infinies des opérations de lecture et d'écriture telles qu'observables à l'interface du système et décrites temporellement et fonctionnellement par les chronogrammes feuilles **the_read** et **the_write**. Le comportement **RAQ**

```
;;; Commande provoquant la generation du modele VHDL
(dump (BEHAVIOR THE_RAQ RAQ (PORT-MAP (SIGNAL INPORT "MESSAGE_TYPE" MESSAGE)
                                      (SIGNAL OUTPORT "MESSAGE_TYPE" MESSAGE)))
      "RAQ"                          ; nom de l'entite a generer
      '("WORK")                      ; librairie VHDL contenant les definitions de types
      '("WORK.RAQ_TYPE.ALL")         ; package VHDL contenant les definitions de types
      '("WORK")                      ; librairie VHDL contenant les definitions procedures
      '("WORK.RAQ_PROCEDURE.ALL")); package VHDL contenant les definitions procedures
```

FIGURE 4.19. Commande provoquant la génération du modèle de la file d'attente

comprend la définition de la variable d'état **QUEUE** et l'attachement de l'appel à la procédure **INIT_QUEUE** à son événement de départ. Ceci permet d'initialiser à vide la variable **QUEUE** au début de l'exécution de ce modèle.

3. SIMULATION DU MODÈLE DE LA QUEUE

Pour simuler un chronogramme, il est nécessaire de passer par les étapes suivantes: génération du chronogramme en VHDL suivie de la compilation du code VHDL. La simulation nécessite également un banc d'essais servant à générer les stimuli à envoyer au système ainsi qu'à récolter ses réponses.

3.1. Génération d'un chronogramme en VHDL. Une fois défini, le comportement **RAQ** doit être instancié pour former un chronogramme qui est ensuite transformé en code VHDL exécutable. Cela est réalisé par l'appel à la fonction LISP **dump** se trouvant dans la figure 4.19. [htbp]

Le premier paramètre est le chronogramme à générer. Le second est une chaîne de caractères représentant le nom de l'entité VHDL à générer. Les quatre derniers paramètres sont optionnels et représentent des listes de chaînes de caractères. La première liste indique quels sont les noms des librairies VHDL contenant des définitions de type à joindre au code généré. La troisième liste est similaire à la première, seulement celle-ci indique les noms des librairies contenant des définitions de procédures et/ou de fonctions. Les deuxième et quatrième listes contiennent les noms des *packages* VHDL à inclure dans le modèle. La deuxième liste indique les noms des *packages* contenant les définitions de type. La quatrième spécifie les noms des *packages* contenant les définitions de procédures et/ou de fonctions.

3.2. Compilation du code VHDL. Le code obtenu par l'appel à la commande dump est directement compilable en VHDL. Pour automatiser la compilation, le scripte **vantage_compile** a été produit par le générateur de code. Celui-ci contient les directives de compilation pour le compilateur VHDL de la compagnie *Vantage*.

Par ailleurs, avant d'exécuter le scripte `vantage_compile`, il est nécessaire que l'utilisateur compile les *packages* contenant les définitions de types et les définitions procédures/fonctions dans les librairies VHDL qu'il a indiquées dans la première et troisième liste de l'appel à la fonction `dump`.

3.3. Le banc d'essais du modèle de la file d'attente. Le modèle de la file d'attente est prêt à être simulé dès qu'est franchie l'étape de sa compilation. Pour exécuter sa simulation, il est nécessaire de disposer d'un environnement permettant de soumettre des stimuli au modèle. Ce genre d'environnement s'appelle un banc d'essais grâce auquel le modélisateur peut explorer le comportement du modèle. Il incombe à l'utilisateur d'écrire le banc d'essais.

Le banc d'essais utilisé pour l'exemple de simulation de la file d'attente génère les stimuli suivants: 3 messages vides sont générés puis déposés sur le port `INPORT` de la file d'attente. Ensuite, 5 messages non-vides sont générés suivis d'encore 3 messages vides. Entre les envois consécutifs de message, il y a un délai de 25 nanosecondes. Le modèle de la queue, lorsque soumis à ces stimuli, devrait rapporter que 5 messages ont été mis dans la file et que 5 messages ont été extraits de la file à chaque 20 nanosecondes d'intervalle. L'utilisateur peut confirmer ces résultats en analysant la trace de la simulation.

Le code représentant l'entité VHDL ainsi que l'architecture de l'entité qui sont utilisées en tant que banc d'essais de la file se trouve en annexe B.

Pour procéder à l'exécution de la simulation du modèle de la file d'attente, il ne reste plus qu'à compiler le banc d'essais précédant puis lancer la simulation.

3.4. Résultat de simulation. Des instructions pour tracer les opérations de la queue ont été introduites dans la définition des procédures `INIT_QUEUE`, `ENQUEUE` et `DEQUEUE`. Ces traces s'ajoutent à celles produites par la librairie d'exécution ADEle pour permettre à l'utilisateur d'analyser le comportement du modèle durant la simulation.

Un extrait de la trace se trouve dans la figure 4.20. La trace complète se trouve en annexe B du mémoire. La trace indique les temps où sont démarrés les chronogrammes. Elle indique aussi quand un événement réel a été jumelé à un événement spécifié ainsi que l'intervalle de temps à l'intérieur duquel le jumelage respecte toujours les contraintes temporelles.

Par ailleurs, la trace met en évidence les temps d'occurence des événements réels lorsqu'ils ont été jumelés aux événements spécifiés de la spécification originale. Les intervalles de temps *assume* et *commit* de ces événements indiquent les plages de temps à l'intérieur desquelles auraient pu être observés les événements réels d'entrée (bornes *assume*) ou les plages de temps à

```
parallel-1/ is started at time : 0 NS
0 NS Initialisation de la queue a vide.
parallel-1/had-loop-1/ is started at time : 0 NS
parallel-1/had-loop-2/ is started at time : 0 NS
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 0 NS
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 0 NS
parallel-1/had-loop-1/leaf-1/ev9626592  occurence time : 15 NS - assume (15 NS, 50 NS)
                                        trigger time : ∞ - commit (∞, ∞).
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 15 NS
parallel-1/had-loop-2/leaf-1/ev9794528  occurence time : 20 NS - assume (0 NS, ∞)
                                        trigger time : 20 NS - commit (20 NS, 20 NS).
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 20 NS
parallel-1/had-loop-1/leaf-1/ev9626592  occurence time : 30 NS - assume (30 NS, 65 NS)
                                        trigger time : ∞ - commit (∞, ∞).
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 30 NS
parallel-1/had-loop-2/leaf-1/ev9794528  occurence time : 40 NS - assume (20 NS, ∞)
                                        trigger time : 40 NS - commit (40 NS, 40 NS).
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 40 NS
parallel-1/had-loop-1/leaf-1/ev9626592  occurence time : 45 NS - assume (45 NS, 80 NS)
                                        trigger time : ∞ - commit (∞, ∞).
45 NS Insertion dans la queue d'un message de valeur => 1
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 45 NS
60 NS Extraction de la queue d'un message de valeur => 1
parallel-1/had-loop-1/leaf-1/ev9626592  occurence time : 60 NS - assume (60 NS, 95 NS)
                                        trigger time : ∞ - commit (∞, ∞).
parallel-1/had-loop-2/leaf-1/ev9794528  occurence time : 60 NS - assume (40 NS, ∞)
                                        trigger time : 60 NS - commit (60 NS, 60 NS).
(...)
```

FIGURE 4.20. Partie de la trace de simulation du modèle de la queue

l'intérieur desquelles auraient pu être générés les événements réels de sortie (bornes *commit*). La trace inclut aussi les temps d'occurence des événements de départ et de fin des chronogrammes.

CHAPITRE 5

Algorithmes et organisation interne de l'outil

Le fonctionnement et l'organisation interne des principaux sous-systèmes de l'outil sont présentés dans ce chapitre. Celui-ci est divisé comme suit: le principe du fonctionnement de l'interpréteur du langage des chronogrammes est exposé, suivi d'une présentation des structures de données LISP servant à représenter les chronogrammes hiérarchiques annotés. L'organisation du code VHDL généré par le générateur de code est ensuite abordée. Finalement, les structures VHDL de données ainsi que l'algorithme de validation des stimuli et de génération des réponses (VSGR) servant à la simulation des chronogrammes hiérarchiques annotés sont détaillés.

1. Interpréteur du langage des chronogrammes hiérarchiques annotés

Le langage reconnu par l'interpréteur est celui décrit par la grammaire en forme normale de Backus contenue en annexe A. L'élaboration de ce langage s'inspire du langage EDIF (Electronic Design Interchange Format) [Ele89] utilisé dans l'industrie des outils de CAO. La réalisation de l'interpréteur s'est faite dans le langage CBDS-lisp [Nor90]. La syntaxe du langage est celle du langage LISP à laquelle sont ajoutées des formes permettant d'exprimer les constituants des chronogrammes hiérarchiques annotés.

1.1. Évaluation des expressions de la grammaire. Avant d'aborder le mécanisme d'évaluation d'expressions du langage des chronogrammes, il est nécessaire de rappeler brièvement comment se fait l'évaluation d'expressions par l'interpréteur LISP.

L'expression suivante exprime dans le langage des chronogrammes une composition conjonctive de deux contraintes de préséance.

```
(CONJUNCTIVE (PRECEDENCE 's1 'p (CMIN 10) (CMAX 20) (INTENT ASSUME))
             (PRECEDENCE 's2 'p (CMIN 5) (CMAX 20) (INTENT ASSUME)))
```

Cette expression est composée de deux sous-expressions :

(i) `(PRECEDENCE 's1 'p (CMIN 10) (CMAX 20) (INTENT ASSUME))`

(ii) `(PRECEDENCE 's2 'p (CMIN 5) (CMAX 20) (INTENT ASSUME))`

Elles mêmes comportent des sous-expressions comme par exemple: `(CMIN 10)`, `(CMAX 20)` et `(INTENT ASSUME)`.

Le langage LISP contient un évaluateur d'expression dont le principe de fonctionnement est expliqué dans ce paragraphe (pour plus de détails voir [Ste90]). Les expressions LISP sont délimitées par une paire de parenthèses. Le premier élément à l'intérieur des parenthèses correspond à un nom de fonction et le reste des éléments aux arguments de la fonction. Les arguments de fonctions sont des objets LISP ou des expressions LISP. Un objet LISP est un élément d'information de base comme par exemple un entier, une chaîne de caractères, une liste, etc. Le résultat de l'évaluation d'une expression est un objet.

L'évaluation d'une expression débute par l'évaluation de ses arguments. Les arguments qui ne sont pas des objets LISP de base mais plutôt des expressions sont évalués de manière à les réduire à des objets LISP. Ainsi, l'interpréteur localise les sous-expressions les plus imbriquées d'une hiérarchie d'expressions, évalue ce niveau et rend le résultat au niveau supérieur. Ensuite, l'évaluation des niveaux supérieurs d'expressions peut débuter et ainsi de suite.

L'idée derière la réalisation de l'interpréteur du langage des chronogrammes est de tirer partie de l'évaluateur d'expression LISP dans l'évaluation des expressions du langage des chronogrammes. Ainsi, pour chaque type d'objets exprimables dans le langage des chronogrammes, une fonction LISP du même nom que le type de l'objet a été définie. Par exemple, voici une expression qui représente une contrainte chronologique de préséance entre l'événement `s1` et l'événement `p`.

```
(PRECEDENCE 's1 'p (CMIN 10) (CMAX 20) (INTENT ASSUME))
```

L'information suivant le nom du type de l'objet (`PRECEDENCE`) représente les attributs de l'objet. Dans cette expression, les attrubuts sont: la source et la cible de la contrainte soit les événements `s1` et `p` respectivement, de même que des sous-expressions qui s'évaluent aux valeurs *min* (10) et *max* (20) de la contrainte ainsi que le genre de la contrainte (`assume`).

La fonction `PRECEDENCE` a pour rôle de s'assurer de la validité syntaxique de l'expression de la contrainte. Elle vérifie donc que les deux premiers arguments sont des noms d'événements et que les deux derniers arguments sont des nombres entiers. Une fois ceci fait, la fonction retourne la représentation interne LISP de cette contrainte de préséance.

1.2. Représentation interne LISP des objets du langage des chronogrammes.

Voici une liste exhaustive des objets exprimables du langage des chronogrammes hiérarchiques annotés ainsi qu'une description des structures de données LISP utilisées pour les représenter.

De manière générale, les objets du langage sont représentés par une liste dont le premier élément est le nom du type de l'objet suivi des attributs de cet objet.

La définition de la représentation interne de chronogrammes est incluse dans les tables 5.1, 5.2 et 5.3. La table 5.1 contient les représentations des chronogrammes feuille, boucle, choix, parallélisation et enchaînement. La table 5.2 détaille quant à elle certaines structures utilisées dans les définitions de la table 5.1 alors que la table 5.3 détaille les représentations des contraintes temporelles.

Dans les définitions de structures de données qui suivent, les conventions typographiques suivantes sont employées:

- les éléments de définitions en caractères `typographiques` représentent des atomes LISP;
- les éléments en caractères *italiques* représentent une instance d'une structure de donnée dont le nom est indiqué;
- la notation (*élément* ...) est utilisée pour indiquer une liste d'objets de type *élément*.

Objet	Type de données
feuille	(`leaf` (*spécification_signal* ...) (*contrainte* ...) *action_départ action_fin*)
boucle	(`had-loop` *appel_prédicat chronogramme action_départ action_fin*)
choix	(`d-choice` ((*appel_prédicat chronogramme*) ...))
parallélisation	(`parallel` (*chronogramme*) *action_départ action_fin*)
enchaînement	(`concatenation` (*chronogramme*) *action_départ action_fin*)
chronogramme	*feuille* ou *boucle* ou *choix* ou *parallélisation* ou *enchaînement*

TABLE 5.1. Représentation interne LISP des chronogrammes hiérarchiques annotés.

2. ORGANISATION DU CODE VHDL ET GÉNÉRATEUR DE CODE

Le générateur de code est le sous-système permettant de traduire la représentation interne LISP de chronogrammes hiérarchiques annotés en un modèle VHDL dont l'algorithme de simulation est décrit dans la section 4.1. Les structures de données nécessaires à l'algorithme de simulation sont générées par ce sous-système. L'une des fonctions du générateur de code est ensuite de construire des fonctions et procédures de support à l'algorithme de simulation. Ces fonctions sont propres à chaque chronogramme en ce qu'elles dépendent des types des signaux utilisés dans ceux-ci, d'où la nécessité de les générer lors de la traduction en VHDL d'un chronogramme. Le code VHDL propre à chaque chronogramme est qualifié de dynamique parce

Objet	Type de données
action_départ	(**start-action** *appel_prédicat appel_procedure*)
action_fin	(**end-action** *appel_prédicat appel_procedure*)
spécification_signal	(**signal-spec** *signal état* (*action_spécifiée* ...))
signal	(**signal** *nom direction type interprétation*)
direction	**in** ou **out** ou **inout** ou **internal**
interprétation	**event** ou **message**
variable	(**variable** *nom type valeur*)
état	*indéfini* ou *constant* ou *valide*
indéfini	(**dont-care**)
constant	(**constant** *valeur*)
valide	(**valid** *variable*)
action_spécifiée	(**spec-action** *nom direction état appel_procédure appel_prédicat*)
appel_procédure	(**procedure-call** *nom variable* ou *signal* ...)
appel_prédicat	(**predicat-call** *nom variable* ou *signal* ...)
intention	**commit** ou **assume**
nom	Atome lisp désignant le nom d'un objet
type	Chaîne de caractère LISP contenant le nom d'un type VHDL.
valeur	Chaîne de caractère LISP contenant la valeur d'un objet.
source, cible	Atome lisp désignant le nom d'un objet action
min, max	Nombre entier

TABLE 5.2. Représentation interne LISP des chronogrammes hiérarchiques annotés (suite 1).

Objet	Type de données
contrainte	*conjonction* ou *premier* ou *dernier* ou *préséance* ou *concomitance*
conjonction	(**conjunctive** *contrainte* ...)
premier	(**earliest** *contrainte* ...)
dernier	(**latest** *contrainte* ...)
préséance	(**precedence** *intention source cible min max*)
concomitance	(**concurence** *intention source cible min*)

TABLE 5.3. Représentation interne LISP des chronogrammes hiérarchiques annotés (suite 2).

qu'il n'est pas fixe et qu'il doit être généré pour chaque chronogramme hiérarchique. Quant au code VHDL commun à tous les chronogrammes, il est plutôt qualifié de code statique. Finalement, pour obtenir l'exécution du modèle, il est nécessaire de compiler les unités générées en plus des unités statiques. Le générateur produit un script en langage de la coquille C de UNIX permettant cette compilation d'unités VHDL par le compilateur VHDL.

2.1. Organisation du code VHDL. La figure 5.21 illustre les dépendances entre les unités VHDL statiques et dynamiques. Les flèches signifient une inter-dépendance entre les définitions contenues dans l'unité à l'origine d'une flèche et celles contenues dans l'unité à son extrémité. Le contenu de chacune de ces unités est détaillé dans les sous-sections suivantes.

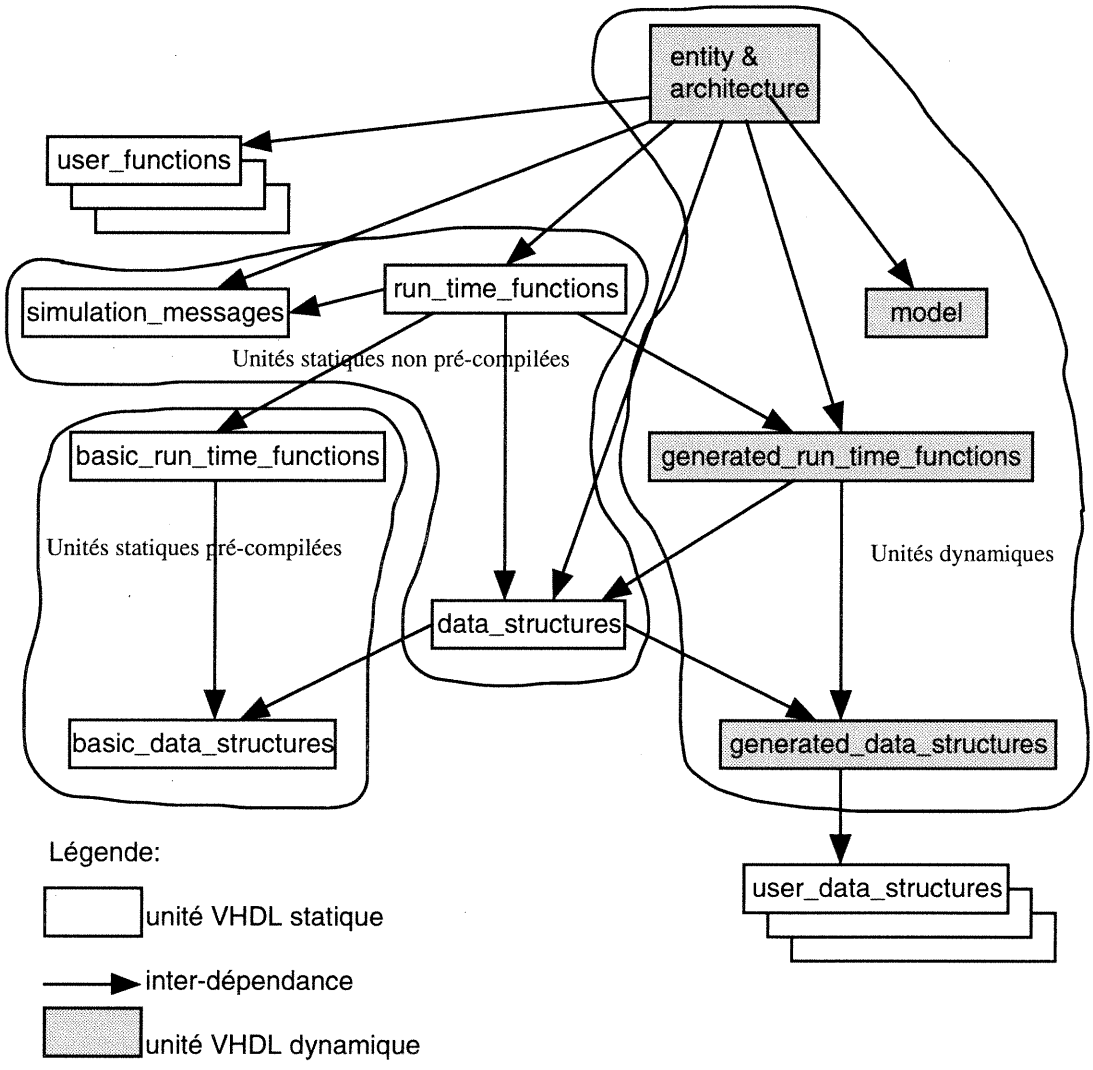


FIGURE 5.21. Interdépendance entre unités statiques et dynamiques en VHDL

2.1.1. *Unités statiques pré-compilées.* Ces unités sont communes à tous les genres de chronogrammes et comme elles ne dépendent d'aucune unité dynamique, elles peuvent être pré-compilées dans une librairie VHDL avant même que ne soient définies les autres unités.

basic_data_structures: Structures de données de base communes à tous les chronogrammes comme par exemple: la définition du type indiquant la direction d'un signal, la définition d'un type *intervalle de temps* pour contenir les temps d'occurence minimal et maximal d'un événement spécifié et aussi la définition d'un type *vecteur de valeurs temporelles* dont les éléments sont des scalaires représentant des unités de temps.

basic_run_time_functions: Fonctions de base définissant les opérations de calcul sur les variables dont le type est défini dans l'unité précédente. Ceci inclut des fonctions comme par exemple: la fonction *maximum* retournant la plus grande valeur scalaire d'un vecteur de valeur temporelle et la fonction *addition* calculant la somme de deux valeurs temporelles.

2.1.2. *Unités dynamiques.* Ces unités sont propres à chaque chronogramme parce qu'elles dépendent des types de ports utilisés dans les chronogrammes. Ces unités sont produites par le générateur de code.

generated_data_structures: Cette unité contient les types de données propres à chaque chronogramme. Elle contient notamment la définition d'une structure de données nommée `uptv` possédant autant de champs qu'il y a de types de ports dans le chronogramme hiérarchique annoté généré.

Cette dernière structure (`uptv`) sert à définir une autre structure représentant les noeuds d'un graphe de contraintes (voir la section 3.2). Les noeuds du graphe représentent quant à eux les événements spécifiés d'un chronogramme feuille tandis que le champ `valeur` de type `uptv` contient la valeur d'une constante s'il s'agit d'un événement spécifié constant.

Le type `uptv` contient un champ pour tous les types utilisés dans le chronogramme et un index pour indiquer lequel de ces champs est désigné pour mémoriser une valeur. Dans l'exemple de la file d'attente du chapitre 4, deux genres de types sont utilisés à l'intérieur des définitions de chronogrammes: le type `MESSAGE_TYPE` et le type `QUEUE_TYPE`. Bien qu'aucun signal ne soit de type `QUEUE_TYPE`, le générateur de code ne tient pas compte de ce fait et considère que les noeuds du graphe de contraintes peuvent contenir des valeurs de ces deux genres de type. Les définitions de type de la figure 5.22 sont alors générées par le système.

```
TYPE type_range IS RANGE 0 TO 1;

TYPE ptr_MESSAGE_TYPE IS ACCESS MESSAGE_TYPE;
TYPE ptr_QUEUE_TYPE IS ACCESS QUEUE_TYPE;

TYPE uptv IS RECORD
   index : type_range;
   type0 : ptr_MESSAGE_TYPE;
   type1 : ptr_QUEUE_TYPE;
END RECORD;
```

FIGURE 5.22. Quelques éléments de l'unité **generated_data_structures**

Le type **type_range** sert à définir le domaine des valeurs possibles du champ **index** du type **uptv**. Les types **ptr_MESSAGE_TYPE** et **ptr_QUEUE_TYPE** définissent des pointeurs sur des structures de données dynamiques de type **MESSAGE_TYPE** et **QUEUE_TYPE** respectivement. C'est d'ailleurs une de ces structures qui contient la valeur à retenir par n'importe quelle variable de type **uptv**. Finalement, le type **uptv** contient un champ de type **type_range** indiquant lequel des champs **type0** ou **type1** retient une valeur. Toute cette mécanique est nécessaire parce que le langage VHDL, de par sa nature fortement typée, ne permet pas de déclarer des graphes dont la définition des sommets est différente. Dans le cas du graphe de contraintes, la différence de type est dûe au fait que les valeurs des constantes associées aux sommets puissent être de n'importe quel type de ports d'un chronogramme.

Par ailleurs, l'unité **generated_data_structures** est produite par le générateur de code, en fonction des types des ports utilisés dans le chronogramme hiérarchique annoté à générer. Le générateur détermine quels sont ces types en effectuant une descente dans la structure de données LISP d'un chronogramme pour y récolter les types des ports du chronogramme.

generated_functions: Cette unité contient la fonction **equal** déterminant l'égalité entre deux variables de type **uptv**. Cette fonction ne peut être décrite qu'une fois la définition du type **uptv** connue. Voilà pourquoi cette fonction est contenue dans une unité dynamique.

model: Dans cette unité se retrouve la procédure **load** permettant de construire la structure de données VHDL représentant le chronogramme généré. Cette structure est détaillée dans la section 3.

entity: Définition de l'entité VHDL représentée par le chronogramme ainsi que l'architecture de cette entité. En VHDL, la déclaration d'entité ne fait qu'établir l'interface du modèle avec son environnement alors que la déclaration d'architecture définit une

réalisation de cette entité. L'architecture contient un processus exécutant l'algorithme de validation des stimuli et de génération des réponses (voir section 4) utilisé pendant la simulation.

2.1.3. *Unités statiques non-précompilées.* Ces unités sont les mêmes pour tous les chronogrammes mais elles nécessitent, pour être compilées, les définitions de type ou de fonction contenues dans les unités générées.

data_structure: Ce sont les définitions de type de données communes à tous les chronogrammes et elles dépendent des définitions de type générées.

run_time_function: Cette unité contient les définitions de procédures et fonctions de support au processus de simulation.

Aux unités définies plus haut viennent s'ajouter les unités de l'utilisateur dans lesquelles sont spécifiées les déclarations de types des ports du chronogramme et les fonctions et procédures utilisées pour l'annotation de chronogrammes. Les unités écrites par l'utilisateur sont les unités `user_data_structures` et `user_functions` dans la figure 5.21. La déclaration de type des sommets des graphes de contraintes dépend de la déclaration du type de données des ports du chronogramme d'où la dépendance de l'unité **generated_data_structures** vis-à-vis les unités de déclaration de types de l'utilisateur dans la figure 5.21.

3. Représentation VHDL de chronogrammes hiérarchiques annotés

Pour représenter un chronogramme hiérarchique annoté dans l'environnement VHDL, il existe une structure de données semblable à la structure requise pour représenter un chronogramme en LISP. La structure VHDL est plus élaborée car elle contient les champs nécessaires à l'algorithme VSGR pour la simulation décrit dans la section 4. Les principales structures de données VHDL sont définies dans l'unité statique non-précompilée `data_structure.vhd` dont la définition en VHDL est donnée en annexe C. Une description moins formelle de ces définitions fait l'objet de cette section.

3.1. Arbre de composition de chronogrammes hiérarchiques annotés. Un chronogramme hiérarchique est représenté par une structure nommée arbre de composition de chronogrammes hiérarchiques annotés. Cette structure correspond à la définition de type `had_node` de l'annexe C. La hiérarchie des noeuds de l'arbre décrit la même hiérarchie que celle du chronogramme. Ainsi, les noeuds de l'arbre représentent aussi des chronogrammes hiérarchiques.

Les noeuds possèdent une des étiquettes suivantes servant à décrire la façon dont sont composés hiérarchiquement les fils de chaque noeud. Les étiquettes possibles sont: `feuille`, `boucle`, `choix`, `parallélisation` et finalement `enchaînement`.

Il va de soi que les feuilles de l'arbre ne peuvent que représenter des chronogrammes feuille.

Lors de la phase de simulation, l'algorithme de validation des stimuli et de génération des réponses (VSGR) fait correspondre les événements spécifiés d'un chronogramme hiérarchique aux événements réels observés. Les noeuds de l'arbre possèdent un champ de type booléen nommé `active` servant à identifier les noeuds de l'arbre menant à des chronogrammes feuille contenant des événements spécifiés susceptibles de correspondre au prochain événement réel à observer ou à générer durant la simulation. Les chemins partant de la racine de l'arbre jusqu'aux feuilles et sur lesquels les noeuds ont le champ `active` à vrai s'appellent les *branches actives* de l'arbre.

Finalement, les feuilles de l'arbre de composition contiennent une liste des ports du chronogramme. Les éléments de cette liste sont des structures nommées `h_port` dans le code VHDL. Les caractéristiques d'un port sont enregistrées dans cette structure. Celles-ci sont principalement:

(i) la valeur initiale du port s'il est initialement à l'état constant;
(ii) un pointeur vers la liste d'actions spécifiées de ce port;
(iii) un pointeur vers la prochaine action spécifiée à faire correspondre avec une action réelle.

Les actions spécifiées correspondent aux noeuds du graphe de contraintes temporelles.

3.2. Graphe de contraintes temporelles. La structure de données retenue pour représenter les contraintes temporelles d'un chronogramme feuille est basée sur celle d'un graphe orienté dans lequel les sommets représentent les événements spécifiés du chronogramme et les arêtes représentent les relations temporelles.

Cette structure de données a pour nom *graphe de contraintes temporelles.*

3.2.1. *Les sommets du graphe de contraintes.* Les sommets du graphe des contraintes d'un chronogramme feuille ont un type de donnée qui correspond au type `spec_action`. Cette structure contient entre autres les informations suivantes :

- le type de l'événement (CONSTANT, VALID, DON'T CARE) que représente le sommet du graphe;
- la valeur de l'événement s'il est de type CONSTANT;

- le temps d'occurence de l'observation de cet événement. Ce champ sert à indiquer à quel moment pendant la simulation cet événement spécifié a été jumelé à un événement réel;
- un identificateur d'appel de prédicat;
- un identificateur d'appel de procédure;
- un identificateur de variable automatique;
- une liste des contraintes temporelles émanant de ce sommet;
- l'arbre de composition des contraintes temporelles de type `assume` dont l'événement est la cible (voir 3.2.4).
- l'arbre de composition des contraintes temporelles de type `commit` dont l'événement est la cible (voir 3.2.4).

Les identificateurs d'appels sont des nombres entiers identifiant un appel de procédure ou de prédicat. Il est nécessaire de recourir à cette technique pour encoder un appel de procédure ou de prédicat car en VHDL, les fonctions et les procédures ne peuvent pas être assignées à des variables du programme.

Similairement, l'identificateur de variable automatique est un nombre entier servant à encoder le nom d'une variable à laquelle la valeur du port sera assignée lorsque l'événement spécifié sera jumelé à un événement réel.

Les arêtes entre les sommets du graphe de contraintes représentent les contraintes temporelles. La liste de contraintes émanant d'un noeud est représentée par une structure de donnée nommée `tc_link` (*timing constraint link*). La structure `tc` est utilisée pour représenter les contraintes de préséance et de concurrence de même que la composition de celles-ci selon les opérateurs `EARLIEST`, `LATEST` et `CONJUNCTIVE`.

3.2.2. *La préséance.* Le premier cas de contrainte à considérer est celui d'une relation temporelle de préséance entre deux événements A (la source) et B (la cible). La relation s'écrit `(PRECEDENCE A B (CMIN u) (CMAX l) (INTENT x))`.

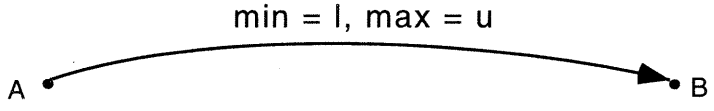


FIGURE 5.23. Représentation de la précédance `(PRECEDENCE A B (CMIN u) (CMAX l) (INTENT x))`

Dans la figure 5.23, les événements A et B sont représentés par les sommets A et B. La contrainte de préséance est représentée par l'arc dont les deux étiquettes sont respectivement

les bornes *min* et *max* de la contrainte. L'*intent* de la contrainte est de type `ASSUME` ou `COMMIT`. Il est à noter que l'*intent* n'est pas indiqué en tant que tel sur l'arc. puisqu'il se déduit plutôt en examinant la cible des contraintes. Chaque sommet contient un ensemble d'arcs `ASSUME` et un ensemble d'arcs `COMMIT` dont il est la cible permettant ainsi de distinguer les contraintes `ASSUME` des contraintes `COMMIT`.

3.2.3. *La concurrence.* La relation de concurrence entre deux événements A et B s'écrit `(CONCURRENCE A B (CMIN l) (CMAX u) (INTENT x))`.

La concurrence engendre deux arcs dans le graphe des contraintes. Le premier a respectivement pour source et cible les événements A et B. L'étiquette min prend la valeur u et l'étiquette max prend la valeur l. Quant au second arc, la cible et la source sont le contraire du premier: soient la source B et la cible A. L'étiquette min a la valeur u et l'étiquette max prend la valeur l.

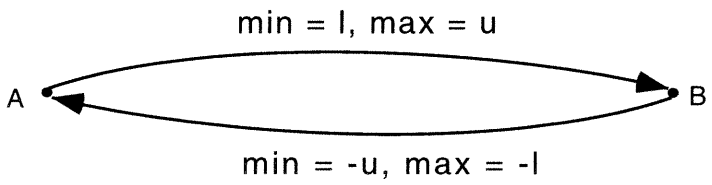


FIGURE 5.24. Représentation d'une concurrence

3.2.4. *Composition de contraintes.* Lorsqu'un événement spécifié est la cible de plusieurs contraintes, celles-ci se composent entre-elles grâce aux opérateurs hiérarchiques `CONJUNCTIVE`, `LATEST` et `EARLIEST`. À chaque sommet sont associés deux arbres de contraintes. L'un représente l'expression de la composition des contraintes `ASSUME` dont il est la cible tandis que l'autre représente l'expression de la composition des contraintes `COMMIT` dont il est la cible.

Les sommets des arbres sont des structures de données de type `tc`. Les sommets feuilles représentent des contraintes de `PRECEDENCE` ou de `CONCURRENCE` tels que décrits dans les deux sous-sections précédentes tandis que les noeuds internes sont étiquetés de l'un des opérateurs de composition (`EARLIEST`, `LATEST` ou `CONJUNCTIVE`). Cette mention indique comment sont composées les contraintes fils de ce noeud.

Il est à noter que les événements qui sont la cible d'une seule contrainte possèdent un arbre de composition comportant un seul sommet. Ce sommet est un pointeur sur l'unique arc représentant la contrainte.

4. Simulation de chronogrammes

Cette section décrit l'algorithme de validation des stimuli et de génération des réponses (VSGR). L'algorithme l'arbre de composition des chronogrammes hiérarchiques annotés de la section 3.2.4 et les graphes de contraintes temporelles de la section 3.2.

L'algorithme VSGR a pour tâche d'associer les événements perçus par le modèle aux événements spécifiés du chronogramme, de vérifier que les événements perçus respectent les contraintes **assumes** du chronogramme et finalement de générer les réponses (les événements de sortie) à l'intérieur de l'intervalle de temps prescrit par les contraintes **commits**.

L'algorithme VSGR est basé sur les travaux de [Duf91] auxquels une modification a été apportée. La modification a pour but d'améliorer l'algorithme de calcul des bornes temporelles d'un événement. Ce calcul détermine quand il est possible qu'un événement spécifié soit observé ou généré. Dans [Duf91], ces bornes s'obtenaient en appliquant un algorithme de calcul du chemin le plus long. Ceci a lieu à chaque fois qu'un événement est observé ou généré. La complexité de l'algorithme du chemin le plus long est de $O(n^3)$ (n étant le nombre de sommets dans le graphe des contraintes).

En contre partie, la procédure de calcul des bornes temporelles décrite dans ce mémoire ne supporte pas la composition *"conjonctives"* de contraintes de type **COMMIT**. Cela signifie qu'elle ne calcule pas correctement quand doit être généré un événement de sortie du système lorsque l'événement est le cible de plus d'une contrainte **COMMIT** et que ces contraintes sont composées entres elles par l'opérateur **CONJUNCTIVE**.

Cette section se divise comme suit. La première section rappelle les grandes lignes de l'algorithme VSGR tel que présenté dans [Duf91]. Les sections suivantes détaillent les parties originales de cet algorithme soit la nouvelle procédure de calcul des bornes temporelles des événements utilisée par l'algorithme VSGR et le traitement d'attachement de procédures, de prédicats et de variables automatiques aux événements spécifiés.

4.1. Algorithme de validation des stimuli et de génération des réponses. Muni des structures de données pour représenter les chronogrammes feuilles et hiérarchiques, l'algorithme de simulation s'acquitte des tâches suivantes :

- valider la valeur et le temps d'occurence (dictés par les contraintes **ASSUMES**) des événements perçus par le système;
- générer les réponses du système dans les délais prescrits par les contraintes **COMMITS**.

L'algorithme présenté dans la figure 5.25 utilise les procédures suivantes:

charge_arbre_de_composition (A): Lorsque cette procédure est appelée, la structure de données représentant le chronogramme hiérarchique annoté à simuler (arbre de composition hiérarchique du chronogramme et les graphes de contraintes des chronogrammes feuilles) est créée en mémoire. `A` est un pointeur sur cette structure de données. Cette procédure est générée par le générateur de code et se trouve dans l'unité dynamique *model*.

La procédure `charge_arbre_de_composition` contient les instructions qui créent dynamiquement en mémoire l'arbre de composition hiérarchique du chronogramme et les graphes de contraintes des chronogrammes feuilles. Ces structures sont implicitement décrites à même les instructions du code.

démarre_chronogramme (A, PROGRÈS, ACTIONS): Cette procédure effectue une descente dans les noeuds actifs (voir section 3 de ce chapitre) de l'arbre de composition hiérarchique pointé par `A`, à la recherche d'un événement de départ qui n'est pas encore survenu. Les événements de départ non-survenus sont les actions de départ des chronogrammes dans lesquels aucune action spécifiée n'a été jumelée avec une action réelle et où aucune action de sortie n'a été générée.

Si un tel événement existe alors il est inséré dans la liste `ACTIONS` et la variable booléenne `PROGRÈS` prend la valeur *vrai*. Dans le cas contraire, la liste `ACTIONS` est retournée vide et la variable `PROGRÈS` prend la valeur *faux*.

Si une action de départ est trouvée, les champs actifs des noeuds fils du noeud de l'arbre contenant l'action sont mis à jour. La mise à jour se fait en fonction de l'opérateur hiérarchique du noeud. Par exemple, dans le cas de l'opérateur *parallélisation*, tous les champs actifs des noeuds fils prendraient la valeur *vrai*.

termine_chronogramme (A, PROGRÈS, ACTIONS): Cette procédure est similaire à la procédure `démarre_chronogramme` sauf qu'elle descend dans les noeuds actifs de l'arbre à la recherche d'un événement de fin qui est prêt à survenir. Une action de fin est prête à survenir quand tous les événements spécifiés distinguables d'un chronogramme ont été jumelés avec des actions réelles et où toutes les actions à générer l'ont été.

Le champ actif du noeud de l'arbre sur lequel un événement fin est prêt à survenir prend alors la valeur *faux*. Par exemple, si l'opérateur hiérarchique du noeud est l'*enchaînement* alors le noeud frère de ce noeud (s'il en existe un) voit son champ actif devenir *vrai*.

calcul_delais_inertie (A, DELTA): Cette procédure descend dans les noeuds actifs de l'arbre de composition `A` pour y rechercher le temps d'occurence du prochain événement à observer ou à générer.

Le temps d'occurence d'un événement à générer est le point médian de l'intervalle de temps `commit` des sommets du graphe de contraintes. Les temps d'occurence des événements spécifiés à observer sont les bornes supérieures des intervalles de temps `assume` des sommets du graphe de contraintes.

La différence entre le temps d'occurence minimal et le temps présent de la simulation est retournée dans la variable `DELTA`. Cette valeur représente le temps d'attente d'un événement d'entrée par le processus exécutant l'algorithme VSGR. À la fin de cette attente l'une des deux propositions suivantes est vraie.

- Un événement de sortie doit être généré; ou alors
- un événement spécifié a atteint la borne maximale de temps à l'intérieur de laquelle il est permis qu'il soit jumelé avec un événement réel. Une contrainte temporelle est violée si l'événement spécifié n'est pas optionnel.

recolte_actions_d'entrée (A, ACTIONS_ENTRÉE): Cette procédure détermine quels sont les événements spécifiés correspondant possiblement aux événements réels observés pendant ce cycle de simulation. Un cycle de simulation de l'algorithme VSGR est une période d'exécution qui se termine quand l'instruction `attendre` est atteinte. (voir figure 5.25). A ce moment, le processus s'endort jusqu'à ce qu'un événement réel survienne à l'interface du système ou que le délai d'attente `DELTA` soit atteint. Après l'une de ces deux conditions, un nouveau cycle de simulation de l'algorithme débute. Les événements spécifiés qu'il est possible de jumeler sont les prochains événements non-survenus sur les ports desquels un événement réel est observé. Les événements spécifiés possibles sont contenus dans la liste `ACTIONS_ENTRÉE` quand cette procédure se termine.

recolte_action_sortie (A, ACTIONS_SORTIE): Cette procédure détermine quels sont les événements spécifiés qu'il est possible de générer au moment de simulation où cette procédure est appelée. Ces événements sont contenus dans la liste `ACTIONS_SORTIE` au retour de cette procédure.

Les événements spécifiés qu'il est possible de générer sont les événements spécifiés contenus dans les branches actives pour lesquels le point médian des bornes temporelles `commit` correspond au temps présent de simulation.

valide_activité (A, ACTIONS_ENTRÉE, ACTIONS_SORTIE): Les listes `ACTIONS_ENTRÉE` et `ACTIONS_SORTIE` contiennent chacune une liste d'événements spécifiés. Cette procédure s'assure que pour chaque événement de ces listes, le temps d'occurence de chacun respecte les contraintes temporelles de type "assume". Si ce n'est pas le cas, le chronogramme contenant l'événement spécifié en question est abandonné en le rendant inactif. Selon le type de l'opérateur hiérarchique du noeud père de celui où est survenu l'abandon, il est possible que l'abandon se propage à ce dernier (dans le cas des opérateurs parallélisation, enchaînement et boucle).

Les bornes d'occurence des événements spécifiés sont mises à jour. Les événements spécifiés dont les bornes doivent être mises à jour sont les prédécesseurs des événements des listes `ACTIONS_ENTRÉE` et `ACTIONS_SORTIE`. Ceci se fait en appliquant la procédure `évaluer_contrainte`. Cette procédure accepte en entrée un arbre de composition de contraintes et retourne en sortie un intervalle de temps à l'intérieur duquel l'action spécifiée peut se produire ou être générée.

appel_prédicats (ACTIONS_ENTRÉE): Cette procédure appelle les prédicats associés aux événements spécifiés passés en entrée dans la liste `ACTIONS_ENTRÉE`. La méthode d'évaluation des prédicats est détaillée dans la section 4.3.

Les prédicats s'évaluant faux causent l'abandon du noeud hiérarchique les contenant. Les événements contenus dans des noeuds abandonnés sont retirés de la liste.

appel_procédures (ACTIONS_ENTRÉE): Cette procédure provoque l'appel des procédures associées aux événements spécifiés contenus dans la liste `ACTIONS_ENTRÉE`.

mise-à-jour_des_variables_automatiques (ACTIONS_ENTRÉE): Cette procédure met à jour les variables automatiques des événements spécifiés contenus dans la liste `ACTIONS_ENTRÉE`

génère_activité (ACTIONS_SORTIE): Cette procédure dépose sur les ports respectifs les valeurs des événements spécifiés contenus dans la liste `ACTIONS_SORTIE`.

Pour plus de détails sur l'algorithme de simulation VSGR, se référer à la citation [Duf91].

4.2. La procédure de mise à jour des bornes d'occurence des événements. La procédure de mise à jour des bornes d'occurence des événements sert à calculer quand il est possible qu'un événement spécifié soit observé ou généré. Cette information est obtenue à partir des contraintes temporelles dont un événement est la cible ainsi que des bornes d'occurence des événements dont ces contraintes sont la source. La mise à jour s'effectue en évaluant la procédure `évaluer_contrainte` .

```
algorithme VSGR

charge_arbre_de_composition (A);
A.actif ← vrai;
PROGRÈS ← vrai;
DELTA ← ∞;
tant que (non A.fini) et PROGRÈS et (non A.erreur) faire
   démarre_chronogramme (A, PROGRÈS, ACTIONS);
   si (non PROGRÈS) alors
      termine_chronogramme (A, PROGRÈS, ACTIONS);
      si (non PROGRÈS) alors
         calcul_delais_inertie (A, DELTA);
         attendre pendant DELTA unités de temps ou qu'un événement réel ait lieu sur un signal d'entrée
         recolte_actions_d'entrée (A, ACTIONS_ENTRÉE);
         recolte_action_sortie (A, ACTIONS_SORTIE);
         pour tout I de ACTIONS_ENTRÉE ∪ ACTIONS_SORTIE faire
             I.survenu ← vrai;
             I.temps_survenu ← temps présent de simulation;
         valide_activité (A, ACTIONS_ENTRÉE, ACTIONS_SORTIE);
         appel_prédicats (ACTIONS_ENTRÉE);
         appel_procédures (ACTIONS_ENTRÉE);
         mise-à-jour_des_variables_automatiques (ACTIONS_ENTRÉE);
         génère_activité (ACTIONS_SORTIE);
         ACTIONS_ENTRÉE ← ∅;
         ACTIONS_SORTIE ← ∅;
fin du tant que
```

FIGURE 5.25. Algorithme VSGR

La procédure **évaluer_contrainte** est appelée deux fois pour chaque événement spécifié des listes **ACTIONS_ENTRÉE** et **ACTIONS_SORTIE** lors de l'exécution de la procédure **valide_activité**. Le premier appel sert à mettre à jour les bornes **assume** d'un événement spécifié. (Ces bornes indiquent entre quels temps de simulation un événement peut être observé). Le second appel met à jour les bornes **commit** d'un événement spécifié. (Ces bornes indiquent entre quels temps de simulation un événement doit être généré).

La procédure **évaluer_contrainte** est présentée dans la figure 5.26.

La procédure **évaluer_contrainte** évalue récursivement un arbre de composition de contraintes pointé par la variable **current**. Si le noeud de l'arbre évalué est une feuille (une contrainte de précédence), les bornes inférieure et supérieure de la variable de sortie **result** peuvent être évaluées. Sinon, selon le type de noeud dont il est question (**EARLIEST**, **LATEST**, **CONJUNCTIVE**), une des procédures **évaluer_earliest**, **évaluer_latest**, **évaluer_conjunctive** est appelée. Le paramètre de l'appel est le fils du noeud **current**. Ces procédures sont détaillées dans la figure 5.27.

L'expression **current.min** représente la borne min de la contrainte **current**. L'expression **current.max** représente la borne max de la contrainte **current**. L'expression **current.source.occ_time** représente le temps d'occurence de l'événement source de la

```
PROCEDURE évaluer_contrainte (VARIABLE current : IN ptr_tc; result : OUT time_interval); IS
  BEGIN
    CASE current.kind IS
      WHEN PREC =>
          IF(current.min < 0 ns) THEN
            result.lower := maximum(0 ns & add(now,current.min));
          ELSE
            result.lower := add (current.source.occ_time, current.min);
          END IF;
          result.upper := add (current.source.occ_time, current.max);
      WHEN EARLIEST => evaluer_earliest (current.son, result);
      WHEN LATEST => evaluer_latest (current.son, result);
      WHEN CONJUNCTIVE => evaluer_conjunctive (current.son, result);
    END CASE;
  END
```

FIGURE 5.26. La procédure **evaluer_contrainte**.

contrainte **current**. Le temps d'occurence est le temps de simulation durant lequel un événement spécifié est jumelé à un événement réel.

Il est à noter que l'événement source de la contrainte **current** peut ne pas être encore survenu. Dans ce cas, son temps d'occurence est considéré comme ayant une valeur infinie. Ainsi, les bornes inférieure et supérieure de l'intervalle **current** peuvent également avoir une valeur infinie.

Les procédures **évaluer_earliest**, **évaluer_latest**, **évaluer_conjunctive** sont semblables. Elles diffèrent seulement dans les calculs des bornes **witness.lower** et **witness.upper** effectués dans le corps de leur boucle. Le reste des procédures est semblable et fonctionne comme ceci: la procédure **évaluer_contrainte** est appliquée récursivement au noeud **current** ainsi qu'à ses frères. La variable **witness** conserve les bornes inférieure et supérieure jusqu'alors calculées pendant l'évaluation des bornes temporelles dans les itérations de la boucle. Finalement, les fonctions **maximum** et **minimum** sont appliquées aux bornes jusqu'à maintenant calculées ainsi que celles du noeud courant.

4.3. Le traitement des annotations. Les annotations que l'algorithme VSGR doit traiter sont les attachements de procédures, de prédicats et de variables automatiques. Les annotations sont associées aux événements spécifiés donc à la structure **spec_action** qui représente les sommets du graphe de contraintes temporelles.

Le traitement des annotations consiste en ceci. Les prédicats doivent être appelés après la phase **valide_activité** de l'algorithme VSGR. Leur évaluation à vrai confirme la validation des événements auxquels ils sont attachés. Les attachements de procédure consistent en des appels à des procédures définies par l'usager. L'appel se fait après l'évaluation du prédicat s'il y

```
PROCEDURE évaluer_earliest (VARIABLE current : IN ptr_tc; result : OUT time_interval;) IS
   BEGIN
      évaluer_contrainte (current, witness);
      cadet := current.brother;
      WHILE cadet /= null LOOP
          évaluer_contrainte (cadet, tmp);
          witness.lower := minimum (witness.lower & tmp.lower);
          witness.upper := minimum (witness.upper & tmp.upper);
      END IF;
      cadet := cadet.brother;
   END LOOP;
   result := witness;
END;

PROCEDURE evaluer_latest (VARIABLE current : IN ptr_tc; result : OUT time_interval;) IS
   BEGIN
      évaluer_contrainte (current, witness);
      cadet := current.brother;
      WHILE cadet /= null LOOP
          évaluer_contrainte (cadet, tmp);
          witness.lower := maximum (witness.lower & tmp.lower);
          witness.upper := maximum (witness.upper & tmp.upper);
      END IF;
      cadet := cadet.brother;
   END LOOP;
   result := witness;
END;

PROCEDURE évaluer_conjunctive (VARIABLE current : IN ptr_tc; result : OUT time_interval;) IS
   BEGIN
      évaluer_contrainte (current, witness);
      cadet := current.brother;
      WHILE cadet /= null LOOP
          évaluer_contrainte (cadet, tmp);
          witness.lower := maximum (witness.lower & tmp.lower);
          witness.upper := minimum (witness.upper & tmp.upper);
      END IF;
      cadet := cadet.brother;
   END LOOP;
   result := witness;
END;
```

FIGURE 5.27. Les procédures `évaluer_earliest`, `évaluer_latest` et `évaluer_conjunctive`.

a lieu et seulement si ce dernier s'avère vrai. Le traitement d'une variable automatique consiste à affecter la valeur d'un port à une variable d'état.

La structure `spec_action` contient 3 champs servant à identifier respectivement l'appel de prédicat, l'appel de procédure et la variable automatique associés au sommet du graphe. (voir section 3.2.1). Les identificateurs sont assignés aux différentes annotations par le générateur de code VHDL. Ce dernier se charge aussi de générer les procédures `appel_action_prédicat`, `appel_action_procédure` et `appel_auto_variable`. Voici une description de chacune de ces procédures.

appel_action_predicat: Cette procédure a pour paramètre d'entrée une liste de sommets du graphe des contraintes et comme paramètres d'entrée/sortie tous les ports utilisés dans le modèle. Ces derniers sont envoyés comme paramètres au cas où les appels de prédicat contenus dans le corps de la procédure aient l'un des ports du modèle comme paramètre formel. La procédure effectue un balayage des éléments de la liste de sommets du graphe des contraintes envoyée dans le paramètre `a`.

```
PROCEDURE appel_action_prédicat (a : INOUT ptr_spec_action; signaux du modèle) IS
  VARIABLE sommet : ptr_spec_action := a;
BEGIN
  WHILE (sommet /= null) LOOP
    CASE sommet.predicate_id IS
      WHEN 0 => NULL;
      WHEN 1 => sommet.pred_call := appel au prédicat correspondant à l'identificateur 1;
      WHEN 2 => sommet.pred_call := appel au prédicat correspondant à l'identificateur 2;
      ...
      WHEN n => sommet.pred_call := appel au prédicat correspondant à l'identificateur n;
    END CASE;
    sommet := sommet.link; – sommet.link pointe vers le prochain sommet à traiter
  END LOOP;
END;
```

La partie de droite des clauses `WHEN` de l'instruction `CASE` enregistre le résultat de l'appel d'un prédicat dans le champ booléen `pred_call` du sommet pointé par la variable `sommet`. L'appel de prédicat correspond à celui qui est retenu dans le champs `predicate_id` du sommet du graphe.

appel_action_procédure: Cette procédure a la même interface que la procédure précédente. Son corps est aussi semblable à celui de la procédure `appel_action_prédicat` bien que l'instruction `CASE` ne soit exécutée que si l'expression `sommet.pred_call` est vraie.

```
PROCEDURE appel_action_procédure (a : INOUT ptr_spec_action; signaux du modèle) IS
  VARIABLE sommet : ptr_spec_action := a;
BEGIN
  WHILE (sommet /= null) LOOP
    IF sommet.pred_call THEN
```

```
        CASE sommet.procédure_id IS
          WHEN 0 => NULL;
          WHEN 1 => appel a la procédure correspondant à l'identificateur 1;
          WHEN 2 => appel a la procédure correspondant à l'identificateur 2;
          ...
          WHEN n => appel a la procédure correspondant à l'identificateur n;
        END CASE;
      END IF;
      sommet := sommet.link; – sommet.link pointe vers le prochain sommet à traiter
  END LOOP;
END;
```

La partie de droite des clauses **WHEN** correspond à l'appel de procédure qu'il faut effectuer quand est terminée la validation de l'événement spécifié représenté par un sommet du graphe.

appel_auto_variable: Cette procédure possède les mêmes paramètres que les procédures précédentes. Son corps est aussi semblable à celui de la procédure précédente.

```
PROCEDURE appel_auto_variable (a : INOUT ptr_spec_action; signaux du modèle) IS
  VARIABLE sommet : ptr_spec_action := a;
BEGIN
  WHILE (sommet /= null) LOOP
    IF sommet.pred_call THEN
      CASE sommet.procédure_id IS
        WHEN 0 => NULL;
        WHEN 1 => variable d'état 1 := référence au port de l'événement
                                       porteur la variable automatique 1;
        WHEN 2 => variable d'état 2 := référence au port de l'événement
                                       porteur la variable automatique 2;
        ...
        WHEN n => variable d'état n := référence au port de l'événement
                                       porteur la variable automatique n;
      END CASE;
    END IF;
    sommet := sommet.link; – sommet.link pointe vers le prochain sommet à traiter
  END LOOP;
END
```

La partie de droite des clauses **WHEN** de l'instruction **CASE** contient une affectation de la valeur d'un port à une variable d'état. La variable d'état est celle annotée à un des n événements spécifiés.

La définition des trois procédures décrites plus haut est faite à l'intérieur de l'architecture VHDL du modèle produit par le générateur de code.

Le traitement des annotations et la nouvelle procédure de calcul des bornes temporelles **évaluer_contrainte** constituent les contributions originales de l'auteur à l'algorithme VSGR.

Maintenant que le fonctionnement et l'organisation interne des principaux sous-systèmes de l'outil ont été présentés, le prochain chapitre résumera les contributions originales de l'auteur en plus de présenter des avenues potentielles de recherches. .

CHAPITRE 6

Conclusion

Ce mémoire présente un outil de spécification pouvant capturer l'aspect temporel et l'aspect fonctionnel des systèmes numériques. L'application de ces spécifications est la vérification de systèmes grâce à la simulation. Un formalisme pour décrire les modèles est proposé (grammaire de chronogrammes fonctionnels). La méthodologie de spécification repose sur une décomposition hiérarchique de sa fonctionnalité telle qu'observée à l'interface du système. L'outil de simulation transforme la représentation hiérarchique du modèle en une forme simulable dans un langage de description de matériel. Le modèle exécutable résultant valide les événements perçus en entrée à son interface, en s'assurant que les événements aient la valeur attendue et que leur temps d'occurence soit celui décrit par les contraintes temporelles de la spécification. Ces dernières dictent aussi quand doivent être produites les réponses (les événements de sortie) du système.

Les contributions originales des travaux rapportés dans ce mémoire sont:

- l'outil de capture de chronogramme feuille Aurora;
- l'élaboration d'un langage permettant d'exprimer les chronogrammes hiérarchiques annotés ainsi que la réalisation d'un interpréteur de ce langage;
- un outil de simulation de chronogrammes hiérarchiques annotés basé sur l'algorithme de validation des stimuli et de génération des réponses.

La méthode de validation des entrées et de génération des réponses (VSGR) utilise les graphes de contraintes temporelles [Duf91]. Les structures de données utilisées demeurent essentiellement les mêmes que dans [Duf91] mais l'algorithme de validation et de génération est plus efficace en terme de complexité de calcul. Malheureusement, l'algorithme ne traîte pas la génération d'événements qui sont la cible de plusieurs contraintes composées par l'opérateur *conjunctive*.

Finalement, une méthode de traitement des annotations (attachement de procédures, de prédicats et de variables automatiques) est réalisée.

Plusieurs travaux futurs sont envisageables. Premièrement, il serait judicieux d'étendre l'algorithme VSGR de manière à ce qu'il traite la génération d'événements qui sont la cible de plusieurs contraintes composées par l'opérateur *conjunctive*.

Le formalisme de description des systèmes temporisés décrit dans ce mémoire est destiné à servir de format intermédiaire entre des outils de capture graphique de la fonctionnalité et des applications consommant ces descriptions. AURORA est l'exemple d'un tel système de capture graphique de diagrammes chronologiques feuilles. Aucun outil n'est disponible présentement pour capturer graphiquement une description hiérarchique de chronogramme. Il faut se rabattre sur la forme textuelle pour les saisir.

Est-il judicieux de capturer l'aspect temporel et d'y ajouter par la suite les annotations fonctionnelles? Cette approche à la modélisation n'est peut-être pas recommandée pour les systèmes dont l'interface est simple mais dont la fonctionnalité est complexe Un plus grand nombre de travaux de modélisation est requis pour déterminer si cette approche est satisfaisante.

La part de ce mémoire consacrée au problème de vérification est uniquement basée sur la simulation. Cette approche est consistante avec ce qui se fait dans l'industrie mais il n'y a aucun doute quant au souhait de disposer de méthodes de vérification formelle. Ce thème pourrait donc constituer une avenue de recherche intéressante pour donner suite à ces travaux.

RÉFÉRENCES

[ABOR90] Airiau (R.), Bergé (J.M.), Olive (V.) et Rouillard (J.). – *VHDL du langage à la modélisation.* – Presse polytechniques et universitaires romandes, 1990.

[ALG91] Antoine (C.) et Le Goff (B). – Timing diagrams for writing and checking logical behavior of integrated systems. *In : Advanced Research Workshop on Correct Hardware Design Methodologies*, pp. 451–455. – Turin, Italie, juin 1991.

[Bor88] Borriello (Gaetano). – *A New Interface Specification Methodology and its Application to Transducer Synthesis.* – Thèse de PhD, University of California Berkeley, 1988.

[Duf91] Dufesne (Mario). – *Un modèle de simulation basé sur les graphes de contraintes hiérarchiques.* – Thèse, Université de Montréal, 1991.

[Ele89] Electronic Industries Association, Washington. – *EDIF Electronic Design Interchange Format Version 2 0 0*, 1989, second édition.

[gro87] group (IEEE Standard 1076-1987). – *IEEE Standard VHDL Language Reference Manual.* – Rapport technique, IEEE, 1987.

[KC93] Khordoc (Karim) et Cerny (Eduard). – Specifying reactive systems with shadow+. – 1993. Document interne au laboratoire de VLSI de l'Université de Montréal.

[KC94] Khordoc (Karim) et Cerny (Eduard). – Modeling cell processing hardware with action diagrams. *In : Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 245–248. – Londre, mai 1994.

[KDC+93] Khordoc (Karim), Dufresne (Mario), Cerny (Eduard), Babkine (Philippe-André) et Silburt (Allan). – Integrating behavior and timing in executable specifications. *In : Conference Proceeding of the IFIP Conference on Hardware Description Languages and their Applications (CHDL)*, éd. par Agnew (David), Claesen (Luc) et Camposano (Raul), pp. 385–402. – Ottawa, avril 1993.

[Nor90] Northern Telecom. – *CBDS-LISP Programmer's guide - volume 1*, 1990.

[SD93] Schlor (Rainer) et Damm (Werner). – Specification and verification of system-level hardware design using timing diagrams. *In : The European Conference on Design Automation with The European Event in ASIC Design*, pp. 518–524. – Paris, février 1993.

[She88] Sherman (S. K.). – Algorithms for timing requirement analysis and generation. *In : 25th DAC.* ACM/IEEE, pp. 724–727. – Anaheim CA, juin 1988.

[Ste90] Steele Jr. (Guy L.). – *COMMON LISP.* – Digital Press, 1990.

ANNEXE A

Grammaire du langage des chronogrammes

La notation suivante est utilisée dans la définition de la grammaire des chronogrammes hiérarchiques annotés en forme normale de Backus.

(i) Les symboles terminaux sont en caractères dactylographique. (`Ce texte est un exemple de type dactylographique.`)

(ii) Une barre verticale "|"separe les formes alternatives.

(iii) Les parenthèses sont utilisées pour mettre en évidence les formes. (ex: $A(A \mid B) \equiv AA$ or AB).

(iv) Les accolades "{ }" signifient zéro ou plusieurs occurences de(s) forme(s) qu'elles englobent.

(v) Les crochets "[ ]" signifient zéro ou une occurence de ls forme qu'elles englobent.

(vi) Les chevrons "< >" englobent une forme qui ne peut apparaitre plus d'une fois a l'intérieur de sa structure syntaxique. (ex. {A | < B >} représente le mot vide et les mots formés de A contenant au plus un B.)

had ::= *had-instance-spec* | *had-body*

had-instance-spec ::= `BEHAVIOR had-instance-nameDef had-type-name-Ref`
{ <*port-map*> | <*param-map*> | <*generic-map*>})

port-map ::= (`PORT-MAP` {*signal-or-port-nameRef*})

param-map ::= (`PARAM-MAP` {*var-or-param-nameRef*})

generic-map ::= (`GENERIC-MAP` {*lisp-expression*})

defbehavior ::= (`DEFBEHAVIOR` *had-type-nameDef* {<*ports*> |
<*parameters*> |
<*generics*> |
signal |
var

<had-body> })

ports ::= (**PORTS** {*port*})

port ::= (**PORT** {*port-nameDef direction v-type-nameRef interpretation*})

direction ::= **IN** | **OUT** | **INOUT**

interpretation ::= **EVENT** | **MESSAGE**

parameters ::= (**PARAMS** {*parameter*})

parameter ::= (**PARAM** *param-nameDef direction v-type-nameRef*)

generics ::= (**GENERICS** {*generic-nameDef*})

signal ::= (**SIGNAL** *signal-nameDef v-type-nameRef interpretation*)

var ::= (**VAR** *var-nameDef v-type-nameRef* [*v-value*])

had-body ::= *leaf* |
had-loop |
concatenation |
parallel |
d-choice |
nd-choice |
exception

leaf ::= (**LEAF** {*carrier-spec* | *constraint* | *<start-action>* | *<end-action>* })

carrier-spec ::= (**CARRIER-SPEC** *signal-or-port-nameRef* { *<initial-spec>* | *action-spec* })

initial-spec ::= (**INITIAL-SPEC** *state* [*action-direction*])

action-direction ::= **IN** | **OUT**

action-spec ::= (**ACTION-SPEC** *action-nameDef state* {*<action-direction-spec>* |
<procedure-call> |
<predicate-call>})

action-direction-spec ::= (**DIRECTION** *action-direction*)

state ::= *dont-care* | *constant* | *valid*

dont-care ::= (**DONT-CARE**)

constant ::= (**CONSTANT** *v-value*)

VALID ::= (**VALID** [*var-nameRef*])

procedure-call ::= (**PROCEDURE-CALL** *v-prog-nameRef*
{*var-or-param-or-signal-or-port-nameRef*})

predicate-call ::= (**PREDICATE-CALL** *v-prog-nameRef*
{*var-or-param-or-signal-or-port-nameRef*})

constraint ::= *conjunctive* | *earliest* | *latest* | *precedence* | *concurrency*

conjunctive ::= (CONJUNCTIVE {<*tc-name-spec*> *constraint*})

earliest ::= (EARLIEST {<*tc-name-spec*> *constraint*})

latest ::= (LATEST {<*tc-name-spec*> *constraint*})

precedence ::= (PRECEDENCE *source-action-nameRef sink-action-nameRef*
{<*tc-name-spec*> | <*intent-spec*> | <*min-spec*> | <*max-spec*> })

concurrency ::= (CONCURRENCY *source-action-nameRef sink-action-nameRef*
{<*tc-name-spec*> | <*intent-spec*> | <*min-spec*> | <*max-spec*> })

tc-name-spec ::= (CNAME *tc-nameDef*)

intent ::= COMMIT | ASSUME | REQUIREMENT

min-spec ::= (CMIN *min*)

max-spec ::= (CMAX *max*)

had-loop ::= (HAD-LOOP *had* {<*start-action*> |
<*predicate-call*> |
<*end-action*> })

concatenation ::= (CONCATENATION { *had* | <*start-action*> |
<*end-action*> })

parallel ::= (PARALLEL { *had* | <*start-action*> | <*end-action*> })

d-choice ::= (D-CHOICE {*choice-branch* | <*start-action*> | <*end-action*>})

nd-choice ::= (ND-CHOICE {*choice-branch* | <*start-action*> | <*end-action*>})

choice-branch ::= (BRANCH *had* [*predicate-call*])

start-action ::= (START-ACTION { <*procedure-call*> | <*predicate-call*> })

end-action ::= (END-ACTION { <*procedure-call*> | <*predicate-call*> })

exception ::= (EXCEPTION {<*condition*> <*normal*> <*handler*>})

condition ::= (CONDITION had)

normal ::= (NORMAL had)

handler ::= (HANDLER had)

ANNEXE B

Code VHDL de l'entité TOP_LEVEL et trace de sa simulation

1. Code VHDL de l'entité TOP_LEVEL

```
entity TOP_LEVEL is
end TOP_LEVEL;

library
  ADEL;
use
  ADEL.RAQ_TYPE.all,
  ADEL.RAQ_PROCEDURE.all;

architecture BEHAVIOR of TOP_LEVEL is
  component RAQ
    port (INPORT : in MESSAGE_TYPE;
          OUTPORT : inout MESSAGE_TYPE);
  end component;

  for all: RAQ use entity ADEL.RAQ(ADELe);
  signal INPORT : MESSAGE_TYPE;
  signal OUTPORT : MESSAGE_TYPE;
begin
  LE_SYSTEM: RAQ
    port map (INPORT => INPORT, OUTPORT => OUTPORT);

  process
  begin
    -- Envoie de 3 messages vides, a 15 ns d'interval
    for i in 1 to 3 loop
       INPORT.ID <= VIDE;
       wait for 15 ns;
    end loop;

    -- Envoie de 5 messages non-vides, a 15 ns d'interval
    for i in 1 to 5 loop
       INPORT.ID <= NON_VIDE;
       INPORT.CHARGE <= i;
       wait for 15 ns;
    end loop;

    -- Envoie de 3 messages vides, a 15 ns d'interval
    for i in 1 to 3 loop
       INPORT.ID <= VIDE;
       wait for 15 ns;
    end loop;
```

```
    – Fin de la sequence de test, attendre indefiniement
    wait;
  end process;
end BEHAVIOR;
```

2. Trace complète de la simulation de l'entité TOP_LEVEL

```
parallel-1/ is started at time : 0 NS
0 NS Initialisation de la queue a vide.
parallel-1/had-loop-1/ is started at time : 0 NS
parallel-1/had-loop-2/ is started at time : 0 NS
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 0 NS
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 0 NS
parallel-1/had-loop-1/leaf-1/ev9626592  occurence time : 15 NS - assume (15 NS, 50 NS)
                                        trigger time : ∞ - commit (∞, ∞).
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 15 NS
parallel-1/had-loop-2/leaf-1/ev9794528  occurence time : 20 NS - assume (0 NS, ∞)
                                        trigger time : 20 NS - commit (20 NS, 20 NS).
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 20 NS
parallel-1/had-loop-1/leaf-1/ev9626592  occurence time : 30 NS - assume (30 NS, 65 NS)
                                        trigger time : ∞ - commit (∞, ∞).
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 30 NS
parallel-1/had-loop-2/leaf-1/ev9794528  occurence time : 40 NS - assume (20 NS, ∞)
                                        trigger time : 40 NS - commit (40 NS, 40 NS).
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 40 NS
parallel-1/had-loop-1/leaf-1/ev9626592  occurence time : 45 NS - assume (45 NS, 80 NS)
                                        trigger time : ∞ - commit (∞, ∞).
45 NS Insertion dans la queue d'un message de valeur => 1
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 45 NS
60 NS Extraction de la queue d'un message de valeur => 1
parallel-1/had-loop-1/leaf-1/ev9626592  occurence time : 60 NS - assume (60 NS, 95 NS)
                                        trigger time : ∞ - commit (∞, ∞).
parallel-1/had-loop-2/leaf-1/ev9794528  occurence time : 60 NS - assume (40 NS, ∞)
                                        trigger time : 60 NS - commit (60 NS, 60 NS).
60 NS Insertion dans la queue d'un message de valeur => 2
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 60 NS
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 60 NS
parallel-1/had-loop-1/leaf-1/ev9626592  occurence time : 75 NS - assume (75 NS, 110 NS)
                                        trigger time : ∞ - commit (∞, ∞).
75 NS Insertion dans la queue d'un message de valeur => 3
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 75 NS
80 NS Extraction de la queue d'un message de valeur => 2
parallel-1/had-loop-2/leaf-1/ev9794528  occurence time : 80 NS - assume (60 NS, ∞)
                                        trigger time : 80 NS - commit (80 NS, 80 NS).
parallel-1/had-loop-2/leaf-1/ is ended at time : ∞
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 80 NS
parallel-1/had-loop-1/leaf-1/ev9626592  occurence time : 90 NS - assume (90 NS, 125 NS)
                                        trigger time : ∞ - commit (∞, ∞).
```

90 NS Insertion dans la queue d'un message de valeur => 4
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 90 NS
100 NS Extraction de la queue d'un message de valeur => 3
parallel-1/had-loop-2/leaf-1/ev9794528 occurence time : 100 NS - assume (80 NS, ∞)
trigger time : 100 NS - commit (100 NS, 100 NS).
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 100 NS
parallel-1/had-loop-1/leaf-1/ev9626592 occurence time : 105 NS - assume (105 NS, 140 NS)
trigger time : ∞ - commit (∞, ∞).
105 NS Insertion dans la queue d'un message de valeur => 5
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 105 NS
120 NS Extraction de la queue d'un message de valeur => 4
parallel-1/had-loop-1/leaf-1/ev9626592 occurence time : 120 NS - assume (120 NS, 155 NS)
trigger time : ∞ - commit (∞, ∞).
parallel-1/had-loop-2/leaf-1/ev9794528 occurence time : 120 NS - assume (100 NS, ∞)
trigger time : 120 NS - commit (120 NS, 120 NS).
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 120 NS
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 120 NS
parallel-1/had-loop-1/leaf-1/ev9626592 occurence time : 135 NS - assume (135 NS, 170 NS)
trigger time : ∞ - commit (∞, ∞).
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 135 NS
140 NS Extraction de la queue d'un message de valeur => 5
parallel-1/had-loop-2/leaf-1/ev9794528 occurence time : 140 NS - assume (120 NS, ∞)
trigger time : 140 NS - commit (140 NS, 140 NS).
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 140 NS
parallel-1/had-loop-1/leaf-1/ev9626592 occurence time : 150 NS - assume (150 NS, 185 NS)
trigger time : ∞ - commit (∞, ∞).
parallel-1/had-loop-1/ loop predicate call returns : TRUE
parallel-1/had-loop-1/leaf-1/ is started at time : 150 NS
parallel-1/had-loop-2/leaf-1/ev9794528 occurence time : 160 NS - assume (140 NS, ∞)
trigger time : 160 NS - commit (160 NS, 160 NS).
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 160 NS
parallel-1/had-loop-2/leaf-1/ev9794528 occurence time : 180 NS - assume (160 NS, ∞)
trigger time : 180 NS - commit (180 NS, 180 NS).
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 180 NS
parallel-1/had-loop-2/leaf-1/ev9794528 occurence time : 200 NS - assume (180 NS, ∞)
trigger time : 200 NS - commit (200 NS, 200 NS).
parallel-1/had-loop-2/ loop predicate call returns : TRUE
parallel-1/had-loop-2/leaf-1/ is started at time : 200 NS
NO MORE STIMULI ARE EXPECTED OR SCHEDULED
SIMULATION STOPS AT TIME : 200 NS

ANNEXE C

Structures de données principales VHDL

```
LIBRARY ADEL;
USE ADEL.BASIC_DATA_STRUCTURES.all, ADEL.RAQ_ADELe_GENERATED_DATA_STRUCTURES.all;

PACKAGE RAQ_ADELe_DATA_STRUCTURES IS
  TYPE tc;

  TYPE ptr_tc IS ACCESS tc;
  TYPE spec_action;
  TYPE ptr_spec_action IS ACCESS spec_action;

  TYPE tc IS RECORD
    kind : constraint_op;
    intent : intend_type;
    min, max : time;
    source, sink : ptr_spec_action;
    name : string (1 to 10);
    son,
    brother : ptr_tc;
  END RECORD;

  TYPE tc_link;
  TYPE ptr_tc_link IS ACCESS tc_link;

  TYPE tc_link IS RECORD
    it : ptr_tc;
    brother : ptr_tc_link;
  END RECORD;

  TYPE h_port;
  TYPE ptr_h_port IS ACCESS h_port;

  TYPE had_node;
  TYPE ptr_had_node IS ACCESS had_node;

  TYPE h_port IS RECORD
    -- Static attributes
    mapping : port_space;
    const : boolean;
    init_value : ptr_uptv;
    first_action : ptr_spec_action;
    brother : ptr_h_port;
    -- Dynamic attribute
    had : ptr_had_node;
    current_action : ptr_spec_action;
```

```
END RECORD;

TYPE spec_action IS RECORD
   – Static attributes
   name : string (1 to 20);
   gendre : action_type;
   direction : direction_type;
   next_action : ptr_spec_action;
   procedure_id : procedure_call_range;
   predicate_id : action_predicate_range;
   auto_var_id : auto_var_assignation_range;
   optional,
   MMR : boolean;
   h_port : ptr_h_port;
   had : ptr_had_node;
   assume : ptr_tc;
   commit : ptr_tc;
   tc_out : ptr_tc_link;
   value : ptr_uptv;
   – Dynamic attributes
   insert,
   pred_call,
   occured : boolean;
   occ_time,
   trigger_time : time;
   a_boundaries,
   c_boundaries : time_interval;
   last_dont_care : boolean;
   – Set attribute
   link : ptr_spec_action;
END RECORD;

TYPE flat_port IS RECORD
   last_observed_activity,
   last_generated_activity : time;
   interpretation : interpretation_type;
   current_value : uptv;
   active : boolean;
END RECORD;

TYPE flat_port_array IS ARRAY (port_space) OF flat_port;

TYPE had_node IS RECORD
   – Static attributes
   kind : had_kind;
   ports_list : ptr_h_port;
   son,
   brother : ptr_had_node;
   loop_pred,
   choice_pred : had_predicate_call_range;
   start_action,
   end_action : ptr_spec_action;
   father : ptr_had_node;
   had_constr_current_int : ptr_constr_current_int;
   – Dynamic attributes
   active,
   disabeled,
   finish,
   error,
   loop_pred_res,
   choice_pred_res : boolean;
```

```
   END RECORD;
END RAQ_ADELe_DATA_STRUCTURES;
```

REMERCIEMENTS

Les travaux rapportés dans ce mémoire n'auraient pu être menés à terme sans le support et la direction d'Éduard Cerny. L'auteur tient donc à le remercier pour sa précieuse collaboration. Doit également être remercié Karim Khordoc pour sa supervision durant la phase de recherche de ce mémoire. Merci à la compagnie *Recherches Bell-Northern Ltée* pour son support tant financier que technique durant l'élaboration de l'éditeur graphique *AURORA*.

D'autres personnes ont contribué à l'aboutissement de la rédaction de ce mémoire. Ainsi, l'auteur tient à exprimer sa gratitude envers Christine Babkine pour ses conseils quand à la rédaction. Finalement, merci à Isabelle Tardif pour son support et sa compagnie durant de longues heures passées à rédiger ce mémoire.

Document Log:

Manuscript Version 1 — le 20 fevrier 1995
Typeset by $\mathcal{AMS}$-LaTeX — 2 July 1996

UN OUTIL POUR LA SPÉCIFICATION DE MATÉRIEL ET LA GÉNÉRATION DE MODÈLES EXÉCUTABLES

PHILIPPE-ANDRÉ BABKINE

Département d'informatique et de recherche opérationelle,
Université de Montréal
CP 6128–A, Montréal (Québec) H3C 3J7, Canada
Tél. : (514) 343-6111 3545#
E-mail address: babkine@iro.umontreal.ca